AD-A104 212 INTERNATIONAL BUSINESS SERVICES INC WASHINGTON DC F/G 9/2
SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP (2ND) AUGUST 21-22, 197--ETC(U)
AUG 78 V R BASILI, E H ELY DAAK70-78-D-0030
UNCLASSIFIED 78CH1390-4C NL

1 OF 3
ADA
104212

SECOND SOFTWARE LIFE CYCLE

AUGUST 21-22, 1978
ATLANTA, GEORGIA

red By: U.S. Army Computer Systems Command
sored By: The IEEE Computer Society

78CH1390-4C

AUGUST 21-22, 1978
ATLANTA, GEORGIA

SECOND SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP

Sponsored By: U.S. Army Computer Systems Command
Cosponsored By: The IEEE Computer Society

78CH1390-4C

PROCEEDINGS PREPARED/DELIVERED BY INTERNATIONAL BUSINESS SERVICES, INC.
UNDER CONTRACT DAAK70-78-D-0030

SECOND SOFTWARE LIFE CYCLE MANAGEMENT

WORKSHOP

20-22 AUGUST 1978

ATLANTA, GEORGIA

SPONSOR

U.S. ARMY INSTITUTE FOR RESEARCH IN
MANAGEMENT INFORMATION AND COMPUTER SCIENCE

313 CALCULATOR BUILDING
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

WORKSHOP CHAIRMAN
VICTOR R. BASILI
UNIVERSITY OF MARYLAND

WORKSHOP DIRECTOR
EDWARD H. ELY
AIRMICS

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Acc

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A104	3. RECIPIENT'S CATALOG NUMBER 212
4. TITLE (~~and Subtitle~~) Second Software Life Cycle Management Workshop August 21-22, 1978, Atlanta, Georgia		~~5.~~ TYPE OF REPORT & PERIOD COVERED (11) Final - August 1978
		6. ~~PERFORMING ORG.~~ REPORT NUMBER (14) 78CH1390-4C
7. AUTHOR(s) (10) Editors: Victor R. Basili University of Maryland and Edward H. Ely AIRMICS (15)		~~8. CONTRACT OR~~ GRANT NUMBER(s) DAAK70-78-D-0030
9. PERFORMING ORGANIZATION NAME AND ADDRESS International Business Services, Inc. 1090 Vermont Ave, NW Suite 1010 Washington, D.C. 20005		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 241
11. CONTROLLING OFFICE NAME AND ADDRESS US Army Institute for Research in Management Information and Computer Science (AIRMICS), 115 O'Keefe Building, GIT, Atlanta, GA 30332		12. REPORT DATE 21-22 August 1978
		13. NUMBER OF PAGES 220
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION A

[illegible]
Appr[illegible] release;
D[illegible] Unl[illegible]

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

Co-sponsored by the IEEE Computer Society

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Life Cycle Workshop, Life Cycle Cost Curves, Management Tools for Software, Software Reliability, Productivity, Budget, Models.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report contains summaries of the sessions conducted at the workshop and position papers presented to the workshop. Session summaries are:

Life Cycle Management Methodology Dynamics - Theory
Life Cycle Management Methodology Dynamics - Practice
Life Cycle Management Measurement Models - Predictive
Life Cycle Management Metrics - Measures & Empirical Studies

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Position papers are:

"Modeling, Measuring & Managing Software Cost"

"Improving the Signal/Noise Ratio of the System Development Process"

"A Step Towards the Obsolescence of Programming"

"A Contingency Theory to Select an Information Requirements Determination Methodology"

"A Life-Cycle Model Based on System Structure"

"The Implications of Life-Cycle Phase Interrelationships for Software Cost Estimating"

"Software Technology and System Integration"

"Establishing a Subjective Prior Distribution for the Application of Life-Cycle Management for Computer Software"

"Design Process Analysis Modeling-An Approach for Improving the System Design Process"

"Life-Cycle Cost Analysis of Instruction--Set Architecture Standardization for Military Computer-Based Systems"

"Useful Evaluation Tools in the Design Process"

"Programmers are too Valuable to be Trusted to Computers"

"Software Cost Modeling: Some Lessons Learned"

"Progress in Modeling the Software Life Cycle in a Phenomenological Way to Obtain Engineering Quality Estimates and Dynamic Control of the Process"

"Software Cost Modeling: Some Lessons Learned"

"A Software Error Detection Model with Applications"

"Laws and Conservation in Large-Program Evolution"

"Validation of a Software Reliability Model"

"Progress in Software Reliability Measurement"

"The Work Breakdown Structure in Software Project Management"

"Operation of the Software Engineering Laboratory"

"Some Distinctions Between the Psychological and Computational Complexity of Software"

"A Review of Software Measurement Studies at General Motors Research Laboratories"

"Software Science--A Progress Report"

"Cost Effectiveness in Software Error Analysis Systems"

"Statistical Techniques for Comparison of Computer Performance"

"Software Complexity Measurement"

"The Utility of Software Quality Metrics in Large-Scale Software Systems Development"

UNCLASSIFIED

"Reliability Evaluation and Management for an Entire Software Life Cycle"

"Analysis of Software Error Model Predictions and Questions of Data Availability"

UNCLASSIFIED

PREFACE

Increasing complexities and challenges of modern systems development have set forth equally new and urgent complexities and challenges relative to advancements in management of computer software life cycles.

As an agency of the United States Army Computer Systems Command, the Army Institute for Research in Management Information and Computer Sciences (AIRMICS) was honored to be able to sponsor the Second Software Life Cycle Management Workshop. This forum brought together some of the most notable contributors within the field of software life cycle management. The collective thoughts of this prestigious group are reflected in these proceedings and should significantly enhance and influence the course of future life cycle management directions.

My sincerest personal appreciation is extended to all those who participated and made the Workshop a highly successful venture in technology exchange.

Clarence Giese
Director, AIRMICS

IN DEDICATION TO THE MEMORY OF OUR COLLEAGUE,

ROBERT McHENRY

TABLE OF CONTENTS

II. EXECUTIVE SUMMARY OF THE SECOND SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP

By Victor R. Basili, Edward H. Ely and Donovan Young

The Second Software Life Cycle Management Workshop brought together 40 Software Life Cycle Management Technology researchers and 90 attendees to discuss theory, practice and technology in managing software over its life cycle.

Building on last year's progress in outlining, identifying and describing the phenomenology of software development, this year's participants discussed progress in validating, refining, extending and exploiting the models and metrics reported in Software Phenomenology (working papers of the Software Life Cycle Management Workshop, August 1977). The main concern expressed by most participants was to foster the emergence of a viable life cycle management technology that could eventually allow accurate estimation and control of time and resources necessary to develop and support software in the military environment.

Topics of interest included (1) description and understanding of various components of the life cycle, (2) ways to delineate and analyze relationships among component activities, (3) milestones and other tools to help direct, coordinate, understand and control research and development in software life cycle management, and (4) development of management tools using the results of life cycle management research to help plan and manage software development projects.

The Workshop was divided into Measurement and Methodology areas, and each of these areas was subdivided as follows:

1. Methodology

   A. Theory: Identification of life cycle components and their interrelationships, based on a comprehensive view of the overall software development process. Chairman, John H. Manley.

   B. Practice: Formal definitions and managment tools found or expected to be useful in life cycle management and control. Chairman, Raymond W. Wolverton.

2. Measurement

   A. Predictive Models: Models derived by analyzing the consequences of a set of assumptions about the development process calibrated by data. Chairman, Lawrence H. Putnam.

   B. Empirical Models: Models based on analysis of data from past and ongoing development projects. Chairman, L. A. Belady.

Participants submitted papers in advance, summing up their research and reporting new unpublished results. These papers appear in the present proceedings.

The two-day workshop began with short presentations by each of the participants to stimulate ideas and discussions. This was followed by a formal discussion session addressing a set of questions (included herein) regarding topics of interest to the Department of the Army. The four groups outlined above met concurrently, each discussing a subset of questions. The results were orally presented by the four Chairmen to the Workshop at large. Summarized versions of workshops findings and recommendations for each of the four topic areas prepared by Session Chairmen are included at the end of this Executive Summary.

A very interesting banquet address was delivered by Mr. E. Larry Dreeman, Chairman of the National Security Team, Federal ADP Reorganization Study, the Presidential Reorganization Project. Mr. Dreeman's remarks centered on the preliminary findings of the study team's efforts and clearly outlined the potential sotfware and related system management challenges facing Department of Defense.

This Workshop attracted substantative papers from several of the most widely-cited investigators in software management. Taken as a whole the papers demonstrate rapid progress in software management science, especially in the field of software cost modeling and prediction (Boehm and Wolverton [129],Elshoff [172], Halstead [174],Nauman and Davis [63],Parr [66], Thibodeau and Dodson [70], Stone and Coleman [93], Velez [95], Putnam [105], and Tausworthe [156]* Less rapid but significant progress is shown in the field of software maintenance and reliability estimation and control (Cur-

*Numbers in brackets refer to page numbers in this document.

tis et al [166], Goel [133], Littlewood [146], Musa [153] Miyamoto [195] and Sukert [209]). Some progress is evident in formal monitoring procedures for software management (Dickover [52], and Basili and Zelkowitz [162]) and in automation of estimation tasks (Shick and Lin [81]), both of which were areas in which many participants expressed intense interest and hopes for further results.

The main conclusion that can be drawn from the collection of papers is that tools for software management and quantitative assessment of the software process are in a state of rapid development. Despite severe difficulties with definitions, taxonomies and data, and despite the fact that the consensus at the end of the 1977 Workshop was that better measurement and definition tools were needed before successful management tools could be developed, the papers tended to concentrate on management tools and measurement rather than definitions. Even the phenomenological papers were written and presented with a clear view toward managerial application. Participants agreed that the technical papers in the Second Workshop were more evaluative and less speculative than those published last year. This was seen as a sign of real progress, but it was also agreed that additional evaluations based on common definitions and real data are still lacking and badly needed.

Summary of Findings

The findings of the Second Workshop include the firm conclusions reported by the discussion sessions and the technical papers, validated by discussion at large among participants and attendees. These findings may be summarized as follows:

1. Formal life-cycle management tools are useful in the development phase (Boehm and Wolverton [129], Parr [66], Stewart [88], Velez [95], Putnam [195]). Management techniques for the development phase of all kinds of software are presented by Dickover [52] and by Naumann and Davis [63]. Many limitations of these methods were discussed; for example, Naumann and Davis cite experience that formal methods are more useful for low-uncertainty projects than for projects involving truly new software.

2. All classes of software should be managed according to a common framework, but different management procedures should be used for each phase of the life cycle. A specific set of life cycle phase definitions was proposed by the Methodology/Theory session group.

3. Overall life-cycle cost curves are useful and promising, but not yet well validated by real data from a multiplicity of environments. Many different sets of assumptions give rise to different mathematical forms of life-cycle curves, all having similar goodness of fit to historical data. Parr [66] and Thibodeau and Dodson [70] offer alternatives to the Rayleigh curve.

4. Reliable data for calibration of life-cycle curves may not be available. In the absence of data-based verification, despite the recommendations of the previous Workshop, a promising alternative is to generate life-cycle curves by aggregating data from PERT analysis (Boehm and Wolverton [129], Parr [66], Tausworthe [156]). This would allow more detailed estimation, and would sidestep the difficulty that empirical data cannot discriminate among alternative aggregated life-cycle curves.

5. Little progress has been made in developing automated management tools for life cycle management of software. Real-time decision-aiding systems would be desirable. Velez [95] has reported an automated way of expressing a target system as a data base in a special language, giving a tangible, measurable object that exists prior to writing any target-system code (details not revealed). Schick and Lin [81] report automation of a small but important task--interactively aiding an expert in the development of subjective probability distribution for a random variable. A PERT-type work-breakdown procedure has been partially automated (Tausworthe [156]).

6. Lines of code per man-month is not a satisfactory indicator of productivity and should be replaced by measures that incorporate quality and complexity as well as length. Reports of successful length-only productivity measurement (Curtis et al [166], Halstead [174] seem to contradict this finding; but their data did not cross organizational boundaries, and Halstead's data came from an organization that enforces standardized coding complexity and corrects line-of-code counts for reuse of standard code.

7. Software reliability modeling is rapidly maturing, so that models such as those of Littlewood [146] and Musa [153] can be used routinely. Elshoff [172] reports success with an (unspecified) predictive measure for estimating the time to revise a program.

8. Good life-cycle models should possess many detailed characteristics such as those proposed by the Methodology/Theory session group.

9. The key goal regarding management information tools is increased visibility by the manager at all times. This implies a local terminal able to report key project aggregates on demand.

10. Software life cycle management problems are people-oriented, not machine-oriented, according to the almost-unanimous consensus of participants.

11. Little or no progress has been made in evaluating maintainability. Effective software life spans have been impossible to estimate in a good or explicit way. Software seems to possess finite life and, thus, must be redone or scrapped within a few years. It appears to be extremely unusual to pay as much as a 10 per cent premium for maintainability.

12. Changes, modifications and enhancements should not be classed and treated as maintenance. Most participants agree, however, that perceived software flexibility (ease of accomplishing prospective modifications) should be considered during design and procurement decisions, and there should a˘so be a recognition during the development phase that changes are inevitable to keep software working while its environment evolves.

Summary of Recommendations

The recommendations of the Second Workshop are the firm recommendations reported by the discussion sessions and the consensus recommendations expressed by the participants and attendees. These recommendations are summarized as follows:

1. Researchers and managers should adopt a standardized set of definitions of terms in software life cycle management. A task force or definitions committee should be organized to work on this problem.

2. Resources should be set aside specifically to validate, classify and test software management models. A project should be initiated to produce an evaluative review or "catalog" of existing descriptive and predictive life-cycle models, describing each model and listing its assumptions, purpose, capabilities and limitations.

3. Research should be done to provide real-time automated management tools for each phase of software life cycle management, using automated metrics and measures that incorporate past project histories and current project information. Automated tools for programmers are also needed.

4. Large-scale data-based validation projects should be undertaken to validate and refine existing and proposed models and metrics and to help provide a basis for standardization of data collection and model parameters.

5. A taxonomy of software environments should be established, and research should be done to elucidate distinctions among individualized software environments. A well-founded taxonomy would allow objective classifications that are indispensable in controlling sources of variance in statistical studies of life cycle phenomena and metrics.

6. Better and more detailed milestone definitions need to be established to provide managers with objective project checkpoints that can be assessed quantitatively.

7. More transfer of technology is needed from project to project and from organization to organization. Intensified effort is needed, not only to provide technology-transfer vehicles such as these Workshops, but also to provide training in methodology and tools for software managers. Effort is also necessary to encourage life cycle management research projects to be carried out in conjunction with ongoing software development and maintenance programs. The Measurement/Predictive discussion group recommends that software project managers not be asked to experiment on their projects, but only to allow data to be collected in a neutral manner. Experiments should make data available to managers to help them manage.

Recommendations that (1) represent minority opinions within the discussion groups or (2) represent personal opinions of the discussion group chairmen are omitted from the above summary but are included in the session summaries which follow.

III. QUESTIONS FOR SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP

The following are a list of questions of interest in the area of Software Life Cycle Management. They are meant to generate discussion and hopefully elicit information of benefit to the community. When considering these questions, keep in mind both the process and the product.

MILESTONES:

Are there better ways to characterize and measure progress than the standard definitions of milestones as points in time? Can we capture the dynamics of the process, i.e., the interactiveness, the user's involvement?

Are there different types of definitions of progress with respect to time and with respect to classes of projects, e.g., first-time efforts, standard developments, etc.?

BUDGET:

What should the people-loading curve be across the life cycle of a system? What effect do different methodologies have on the shape of that curve? How do you decide how much do to do within a fixed budget? What is the effect of size and organization on the budget? What generic factors are/should be present in software cost-estimating formulas? What are the techniques for measuring work accomplished versus budgeted dollars?

PRODUCTIVITY:

Are there better ways to measure group productivity than lines of code per man month? How can one measure individual productivity or productivity on small projects? What are the important factors for measuring productivity during development, during modification?

TOOLS:

Where should we be going in terms of automated tools for managing the software development process and aiding the development personnel (Management information vs. product generating tools)? What should tools encourage?

EMPIRICAL STUDIES:

What do we really need to know to understand the process better? What information should be collected about the process, the product and their interaction, and for what purpose? What kinds of experiments and evaluations should be performed? How can we capture the idea of program complexity? How can program managers be convinced to conduct experiments on their programs? What progress, if any, is being made on the transfer of learning from project to project within and between organizations?

MODELS:

Give a set of criteria for good predictive models of the software life cycle. How could statistical and analytical models be combined? Is there a need for a "standard" set of generic models of the software life cycle process?

METHODOLOGIES:

What are the components of an overall methodology? Where should software technology be going? Are there "standardizable" methodologies? What effect do different software development characteristics have on the implementation or adaption of methodologies in practice? How do you characterize a methodology? How many methodologies exist in practice?

MAINTENANCE/MODIFICATION:

Is there a way to determine and measure the effective life span for software systems? How do you know when to redo a system? What are the design trade-offs for maintainability? At what point in the life cycle should maintenance considerations be included? What strategy should be used to transfer software from developer to maintainer?

MANAGEMENT:

What are the major ingredients in the management of software? What makes it unique? What makes it different from hardware, for example? How should the organizational structure relate to the problem to be solved and the different interactive phases of development? To what extent should managers be technically trained/involved? To what extent should technical personnel be managerially trained/involved? Are there different classifications of software that require different methods of management (e.g., embedded vs. non-embedded)? What are they? Are there predictable crises in the software life cycle and what are the early warnings?

ENVIRONMENT:

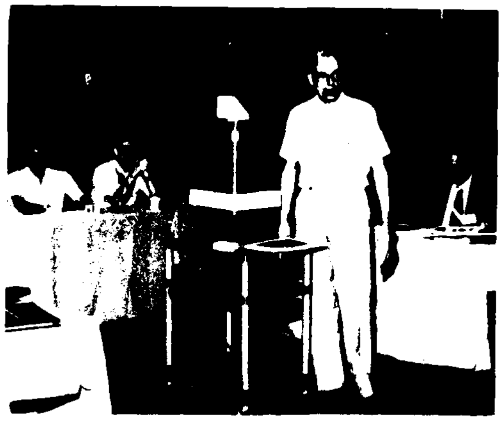
Is there anything different in the above with respect to future technological developments, e.g., small computer environments, standardized modules? What changes need to be made? Is there any "scaling" effect?

GENERAL:

What parts of the above questions should be attacked first? What are solvable in the next five years? What questions would you like to see asked next year? What should research efforts concentrate on? What are the ten top software life cycle management terms that need definition? What is your source authority for present definitions, e.g., ANSI?

...workshop in progress...

Dr. Clarence Giese (right), Director of AIRMICS, gave the Welcoming Address


Dr. Giese in progress.

Mr. Lawrence Putnam (2d from L.) chairs session on Predictive Measurement Models

Mr. E. Larry Dreeman (right), Chairman of the National Security Team, chats with attendees after his banquet speech

Attendees in General Session.

IV. SESSION SUMMARIES

I. LIFE CYCLE MANAGEMENT METHODOLOGY DYNAMICS & THEORY

Chairman: Dr. John H. Manley, Johns Hopkins University

PANELISTS

John R. Brown	Harvey Koch
Thomas DeLutis	J. David Nauman
Melvin E. Dickover	Francis N. Parr

Robert Thibodeau

SOFTWARE LIFE CYCLE MANAGEMENT: DYNAMICS THEORY

Summarized by

Dr. John H. Manley

The Johns Hopkins University
Applied Physics Laboratory
Laurel, Maryland

Abstract

The Dynamics Theory Group discussed the conceptual relationships of software to a system life cycle model, and management to a composite software life cycle model. A "standard" software life cycle management model is proposed that is a modification of the generally accepted Department of Defense system life cycle model. The group agreed that there are at least five distinct types of Army management involved in a major system software management life cycle, and linked quantitative life cycle milestones to elementary decision theory. Other findings are reported that suggest several profitable areas for Army software management research. A summary of a separate report being submitted to the Chairman of the President's ADP Reorganization Project National Security Team is included as a sequel to the highly stimulating Monday night banquet presentation. The report recommends that ADP resource management policymaking for technical issues be centralized; ADP resource acquisition management control be functionally decentralized, and; a systems management approach similar to that developed for embedded computer systems be used for ADP system life cycle management.

Introduction

The Dynamics Theory Group focused on the theoretical aspects of software life cycle management (SLCM). We interpreted our workshop charter quite literally and decided not to get involved with the details of management practice or tools since it appeared during the opening plenary session that the other three groups would adequately cover those aspects of SLCM. Therefore, we tried to answer only a few of the most basic questions that were central to our "dynamics theory" view of the world. For example, we tried to determine whether or not traditional ways of modeling the software life cycle, to include milestoning, could be improved upon.

We also learned from the plenary session that the other three groups intended to concentrate on the Full-Scale Development phase of the Department of Defense (DoD) system life cycle which is primarily involved with the program management aspects of developing software. Therefore, the predominant activity addressed by our group was management of the complete software life cycle, with emphasis on aspects other than the more popular area of program management of the software development process. It was our feeling that the Army should be concerned with a wider range of management problems to include:

a. Developing better ADP and tactical system requirements that include software as component parts.

b. Administering (as the buyer) the software development technical management process which in many cases is carried out by outside contractor organizations.

c. Developing a better understanding of how to economically maintain software contained in operational systems, both as a user and logistician.

Thus, our deliberations were directed more from the perspective of Army managers, be they corporate, field, technical, program, or logistics. We then took our first step into the arena of software life cycle management methodology theory by trying to identify life cycle components and their relationships based upon this Army manager perspective of the software development process.

SLCM Management Differences Fact or Fancy?

The first specific question we addressed was: Are there different classifications of software (for example, embedded versus non-embedded or functional versus non-functional) that require different methods of management and, if this is true, what are they?

We reached the unanimous conclusion that:

> Appropriate management methods that are applicable to different phases of the life cycle do not vary across different classes of software, but, the specific management method used can be and usually is different for different phases of the software life cycle.

This first finding prompted us to closely examine the life cycle management process using a somewhat different perspective than has been customary in the past. As a first step, we found that we could use the Department of Defense (DoD) system life cycle as it is commonly understood (Reference 1) to generally fit our individual theories of what software life cycle management means. However, we were forced to make what turned out to be several conceptually significant modifications to that "system" model in order to develop a useful "software" life cycle management working model.

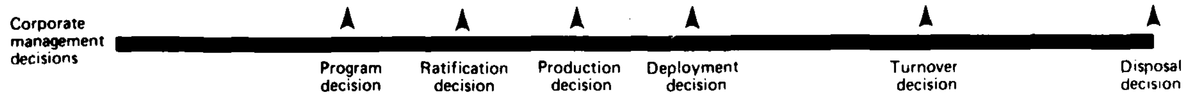
We recommend that our proposed software life cycle management model shown in Figure 1 be used as a strawman baseline for a follow-on research effort by the Army to add conceptual detail to the individual life cycle subphases. Since the top levels of life cycle phase terminology used in existing Department of Defense documentation remain the same, that is, Conceptual, Validation, Full-Scale Development, Production, Deployment and Support phases, we do not advocate replacing any existing documentation but would simply modify it as described below.

Conceptual Phase

As shown in Figure 1, the Conceptual Phase has been divided into two subphases, Conceptual Requirements Definition and Conceptual Requirements Validation.

Definition Subphase. Most front end major system requirements development activity is performed by DoD functional user or field organizations for major systems. In fact, field commands usually employ relatively large groups of people (development planners) who perform this type of analysis on a continuing basis. Since they necessarily possess almost a purely functional systems orientation, any ADPE (Automatic Data Processing Equipment)

Defense system life cycle major phase	Conceptual		Validation	Full-scale development	Production	Deployment		Support	
Software life cycle subphase	Requirements definition	Requirements validation	Validation	Full-scale development	Production	Debugging	Fine tuning	Maintenance	Modification



Type of management

Functional or line	Primary	Advisory	Advisory	Advisory	Advisory	Advisory	Advisory		Advisory
Technical (new technology)		Primary	Advisory	Advisory				Advisory	Advisory
Program			Primary	Primary	Primary	Primary	Primary		Primary
Logistics			Advisory	Advisory	Advisory	Advisory	Advisory	Primary	Advisory

Fig. 1 Software life cycle management model.

included in such systems is almost always embedded in the classic sense of the original definition, that is:

> "...a computer can be considered to be of the embedded variety when it is:
>
> 1. physically incorporated into a larger system whose primary function is not data processing; and
>
> 2. integral to such a system from a design, procurement and operations viewpoint." (Reference 2).

It is important to note that such work does not usually involve software requirements analysis at this, the very front end of the greater system life cycle subcycle, but nevertheless always represents the inital activity on first-time systems.

The term "greater system" will be referred to subsequently and requires clarification. In the context of embedded computer resources in the tactical or defense systems world, the greater system refers to the tank, aircraft or missile that contains the embedded computers, computer programs and computer data as component parts. We will show later that this same interpretation can also be conceptually applied to greater (data processing) systems such as supply, personnel, transportation, payroll and so forth.

Validation Subphase. The Conceptual Requirements Validation subphase comes next. It is distinct from the preceeding subphase in that additional players are involved during the important paper feasibility study activities. When deemed necessary (always for major defense systems) these feasibility studies are usually carried out jointly by both the using (field) and development commands. The development commands are three of the Joint Logistic Commands (JLC) consisting of the Army's Materiel Development and Readiness Command (DARCOM), the Air Force Systems Command (AFSC) and the Naval Material Command (NMC or NAVMAT). These conceptual system feasibility studies seldom involve outside contractors and usually have high military security protection, thus being quite invisible to both the academic and commercial software communities.

Thus, we see the conceptual phase as necessarily consisting of two distinct subphases, Conceptual Requirements Definition and Conceptual Requirements Validation. As will be explained later, the changes in the participant mix from the first subphase to the next has a direct bearing on management of the software life cycle.

Validation Phase

The Validation Phase is essentially one in which the DoD validates the solution to the previously validated greater system requirement. This is the first phase where software is generally acknowledged explicitly as a system resource. Specifically, in this phase the program characteristics of performance, cost and schedule are validated and refined through extensive study and analysis, actual hardware development, or possibly prototype testing. The main idea here is that hardware development and evaluation may provide corporate management (Service Staff, Office of the Secretary of Defense and the Congress) with a better definition of program characteristics, higher confidence that risks have been resolved or minimized, and a greater confidence in the ultimate outcome than could the paper studies generated during the preceeding Conceptual Phase.

In this second major phase, an intial Program Office cadre is expanded to a full program office and, if a major system is involved, software becomes an item of specific interest as directed by Secretary of Defense level instructions (for example, see Reference 3).

Full-Scale Development Phase

During the next major life cycle phase, the system including all of its support items is designed, fabricated and tested. The intended output is, as a minimum, a preproduction system that closely approximates the final product, the documentation necessary to enter the Production Phase, and the test results that meet requirements. Since software can be replicated precisely, this phase is the most important with respect to the development of quality software, and where most of the emphasis has been placed to date (as opposed to hardware where production problems can be severe and very costly).

Production Phase

When more than one copy of a system must be produced, this phase becomes important, especially if production of copies is to be carried out over a period of years as in the case of major defense systems. One of the most difficult areas of management in this phase involves change control, with its difficulty being directly proportional to the amount of change activity. The same methods of planning, development and testing should be followed to make system changes during production runs as were used previously in the Full-Scale Development phase.

When producing copies of software systems for multiple users, the same principles apply and a quite close analogy to hardware methods can be conceptualized. Latent defects will emerge during this phase and the user will continue to require changes to the system, both in hardware and software. Each change must be treated as a mini full-scale development, both in principle and in fact, especially with regard to management methods.

Deployment Phase

This phase has been interpreted and defined somewhat differently by each of the three military services and is consequently difficult to describe in terms of a "standard" software life cycle model. In general, this phase includes events generated when final copies of new systems are actually put into operational use in field organizations. We feel that additional emphasis should be placed on the management activities in this phase, especially with respect to the problems of transfer of development management responsibility to the support management team. It is an extremely difficult problem to determine precisely the point when any system (or copy thereof) becomes truly "operational." We recommend that the Army follow up on this point with further research to develop an effective system (software) deployment strategy. To provide a starting point, we offer the following description of two proper subphases which we feel must be included as a minimum.

Debugging Subphase. At the beginning of the Deployment Phase when a new system is produced and first handed over to the using organization, the embedded software will necessarily contain latent defects. This means that the initial activity in the phase will involve bug removal, or simply trying to get the system to run smoothly in the user's environment. We call this the Debugging subphase. This does not mean, however, that when most bugs are removed, the system will fully meet current user requirements, even though it has been proven to meet the original systems specification through extensive testing.

Fine Tuning Subphase. Thus, a second distinct subphase we call Fine Tuning is necessary. This involves tailoring the system to meet current user requirements. When one reflects on the length of time involved for systems to pass through the Full-Scale Development and Production phases, it is small wonder that they usually do not meet "current" user's requirements that have evolved over a period of perhaps 2, 5 or 10 years. Only when this fine tuning is completed however, should the system be allowed to enter what we generally understand as the Support Phase. But, how does one determine when this activity is truly finished? Again, we recommend the Army focus a significant research effort on this critical problem area.

In any event, the end point of the Deployment Phase is quite clear, the system and all of its associated software has been "thrown over the wall" to the logistics or support manager...ready or not.

Support Phase

As has been reported in this workshop and elsewhere, the Support Phase is now regarded as the highest cost phase with respect to the total software life cycle. It is the one on which we need to focus most of our attention if we ever hope to significantly reduce software life cycle costs. This is also the phase in which most of the ADP or non-embedded computer system action resides. In short, the Support Phase is where most of our money is spent and where the Army should concentrate its research efforts at finding ways to reduce software support costs.

In view of its importance, the Dynamics Theory Group spent most of its time discussing this phase and its software life cycle management implications. We began by defining what activities should be carried out in the Support Phase. We decided that software support should include at least the following continuing tasks:

a. Continuing correction of latent bugs and technical deficiencies in software as they are discovered.

b. Making system changes due to modifications to equipment (hardware) that fails, wears out or is replaced for other technical reasons.

c. Any other external impact forcing a software change that does not affect user greater system functions.

These activities are what we consider to constitute normal software maintenance in a quite general context.

Modifications, however, will also occur for other reasons. The main one being that the user requires a change to the functional characteristics of his greater system. If such a modification involves changing software (or hardware), a quite different type of support activity will be required.

Once the support manager begins to invoke this last type of change process, he is changing more than the data processing components and, unfortunately, this fact is not always recognized in the professional community. Whenever the greater system is impacted, the life cycle begins a new subcycle that is very similar to jumping back to the Full-Scale Development phase and possibly to an even earlier phase. Since this means that we must reapply project management methods through the modification development, debugging and fine tuning processes, we have defined this continuing cyclical activity as a separate subphase called System Modification.

System Disposal

Eventually, whether it be six months or fifty years, when corporate management decides that the greater system is no longer needed by the organization, it is removed from the active system inventory, thus ending the software management life cycle.

Management Relationships to the Modified Software Life Cycle

As shown in Figure 1, five distinct types of management are involved during the different phases and subphases of the life cycle. Their relationships to the phases and also to each other are briefly described below.

Conceptual Phase Management

From this very first life cycle phase through to the ultimate disposal of the system, the functional manager is (or should be) involved. He is the first one to organize a team to develop the initial greater system requirements. He oversees his system during its development, receives it in the field and operates it until its disposal. This type of management is primarily concerned with how the greater system performs its military mission. All computer resources are treated as embedded, that is, as component parts of the greater system. The functional manager's continuing bottom line activity is to plan, develop and employ systems as necessary to achieve his operational objectives.

Whenever a new system or major system modification concept defined by the functional manager includes future or even state-of-the-art technology, scientific and engineering people are usually employed in the Requirements Concept Validation subphase to perform technical feasibility studies, and later to develop any required new technology. It is generally recognized that day-to-day management of scientists and engineers is quite different than so called "line management" of personnel who operate greater systems in the field. There are, in fact, different curricula in most major universities that address engineering management/administration separately from business management/administration. Thus, it must be recognized that in the early stages of the software life cycle, technical management as a subset of general management is often involved.

Validation Phase Management

Once the system begins the Validation Phase, planning for system development begins in anticipation of an approval for system full-scale development and eventual production. It is here that the type of program office is defined, the determination of the type of contracts to be used, decisions on which components will be developed in-house and which out-house, the type and level of expertise the program manager should have and so on. Once these preliminary matters are determined by corporate management, a program manager is hired and a program office cadre assembled. This is the point when program management as a separate type is begun as an addition to the existing functional area management and technical management teams already on board.

The distinguishing feature of program management as a type is that the program manager must live within three basic constraints:

a. A "fixed" budget.

b. An "inflexible" schedule.

c. A "constantly changing" specification of the greater system he is chartered to build.

This type of management is the one we generally talk about at workshops and conferences such as this, usually to the exclusion of the other types shown in the expanded life cycle model. This is not meant to imply that the program management for a new system development or major modification to an existing system is not a most critical occupation. However, our group recognized that this is definitely not the only type of management involved in the overall software life cycle.

Full-Scale Development Phase Management

Sometime during the development phase, logistics people must get involved to insure that the greater systems being developed will be delivered as "maintainable" during the operations phase. Once the logisticians get on board as shown in Figure 1, they remain involved with the system until its ultimate disposal.

Production/Support Phase Management

When the system has been developed to the point that the major bugs have been removed and it has been fine tuned to satisfy the user, responsibility for the system is formally transferred from the program manager to the logistics manager and the user for the remainder of the life cycle. As mentioned before, the system can be changed (sometimes quite drastically) by the logistics manager with no outward appearance of change to the user in functional characteristics. When the user requires a change, however, both the user and the logistician must work together with a project manager to eventually develop a new system, that is, System XYZ-Model 2, -Model 3, -Model 4 or -Model n.

Notice that we make the distinction between "program" and "project" managers. The program manager is the one involved with the original development of the greater system. A succession of project managers are those responsible for making modifications to the delivered greater system during the Support Phase of the system life cycle. However, both program and project managers use the same tools and techniques to perform their important functions.

Corporate Software Life Cycle Management

Of course, overseeing all of the management types mentioned thus far are the top-level decision makers involved with deciding: (a) when the system can proceed from one major phase to the next (Defense System Acquisition Review Council), (b) whether or not the system will be funded from year to year (Congress), (c) whether a particular computer will be approved for purchase (General Services Administration), (d) whether the system is needed to fulfill greater National security needs (President, Secretary of Defense, Service Secretaries, Joint Chiefs of Staff and the Service Staffs) and so forth. We consolidated all of these top-level managers under the global title of corporate management.

Army Management Implications

The implications for Army management that stem from our life cycle model and the concomitant management relationships described above are contained in this section of our report. Our discussions in developing the modified software management life cycle model generated many digressions, some of which resulted in theoretical conclusions that we felt warranted reporting, while others did not. Thus, the following topics are offered as a selection of those we feel are worth consideration by the Army as potential topics for further research.

Life Cycle Cost Implications

Upon close examination of a variety of Army and other automated systems, we found that we could easily fit any kind of a greater system into the modified life cycle model described above. Clearly, major defense systems such as missiles, tanks, aircraft, command and control systems, ships or the like that contain embedded computer resources pass through every phase we have described. For example, one speaker commented in the opening plenary session that the B-52 bomber's original conceptional requirement was developed in approximately 1948 and the system deployed in approximately 1952. Since that time, it has undergone many major modifications. He stated quite accurately that it is highly unlikely that anyone at the beginning of that program could have foreseen that the B-52 would still be flying today, 30 years after system concept definition.

The management implication of such long term continuing modifications to a major system is that their cost over the life cycle cannot be planned or even speculated with any degree of reasonable accuracy. Thus, our group concluded that:

The planned cost of software embedded in a system should terminate with the initial entry of the system into the System Modification subphase of the Support Phase of its life cycle.

From that point on, each new modification should be treated as a separate project to be independently managed over a mini-life cycle.

Another example closer to the Automatic Data Processing community is the automated post or base supply system that is common throughout the Department of Defense. The Conceptual Phase for the automated supply system began in the 1950's and the system became fully operational in the 1960's. Since then, the system has been in the Support Phase. This perspective of that greater system, that is, the automated base supply system, indicates that it may never reach the Disposal milestone. Thus, we have had for the past decade a succession of both maintenance actions and system modifications which have been successfully carried out as they became necessary. In no case, however, has a totally new supply system emerged which required starting back at the beginning of a complete greater system life cycle.

Embedded Computer Resource Management Implications

Our finding above that the supply system is conceptually the same as the B-52 with respect to its embedded computer resources has another implication. As some of the original greater system embedded computers wear out and have to be replaced, they fall under the Maintenance subphase. If the original software is captured through emulation on replacement machines, we have not really changed the system in the eyes of the functional user. However, if we make a modification to satisfy new or emerging user requirements, then we go through the project management and the development life cycle as far back as the change warrants.

Upon closer examination of the differences between so-called embedded computer systems and non-embedded or general purpose automatic data processing, we found that the basic distinctions between these two categories were as follows:

a. There is a definite difference in procurement methodology in the DoD between embedded computers and commercially available ADPE. In fact, this area of of such great importance it will be discussed separately as the next major management implication titled "Procurement Management Implications."

b. There is another major difference between embedded and non-embedded computer systems with respect to management. This is that the embedded computer system is normally involved in the complete life cycle model as developed by our group. Most ADP management efforts are only involved with the Support Phase of the life cycle, since major ADP systems such as supply, personnel, finance, inventory control and so forth were originally developed 10 to 20 years ago. Current software development programs are either of the maintenance or modification variety and do not, in most cases, involve new functional systems that begin in the Conceptual Phase.

Procurement Management Implications

We stated above that the methodology for procuring Automatic Data Processing Equipment (ADPE) is different than that used for procuring Embedded Computer Systems (ECS) in the Department of Defense. Thus, the answers to the questions of whether or not ECS or ADP systems require different types of management practice or different types of life cycles are simply yes and no, respectively.

The distinction that has been made between these two categories of automated systems was a direct result of differences in procurement regulations stemming from the 1965 "Brooks Act" (Reference 4). The Armed Services Procurement Regulation (ASPR) is used to procure defense systems and, in most cases, any computers embedded in such systems. These are excluded from the ADPE procurement regulations that implement Brooks Act guidelines, as administered by the Office of Management and Budget (OMB) and the General Services Administration (GSA).

Implementation of instructions concerning "GSA-controlled" ADP computers on one hand, and "excluded" embedded computers on the other, has resulted in two separate series of directives and instructions at the Office of the Secretary of Defense level. The ADPE or controlled computer procurements fall under the purview of the Assistant Secretary of Defense (Controller) and are controlled under the "4000 Series" instructions. The excluded embedded computers fall under the jurisdiction of the Office of the Undersecretary of

Defense for Research and Engineering. They use the ASPR for most ADPE procurement actions as governed by "5000 Series" instructions.

To complicate matters further, each service has developed its own unique method for implementing the 4000 and 5000 Series regulations quite independently of one another. Hence, the answer to the original question is yes, embedded computer systems (ECS) and ADP systems do require different types of management, but not different types of life cycles.

This creates a problem of qualification for the managers involved in the life cycle of the system. In no way, however, does this change the model presented as the procurement method is only a policy guideline for management to follow.

Functional Manager Involvement

Now what did we learn from an examination of this revised life cycle model? First, we found that functional managers must stay involved in the software aspects of their greater systems throughout the life cycle which, as we have seen, can be an extremely long time. They cannot abrogate this responsibility. The implication here is that there must be an office of primary responsibility established for that system and the responsibility carefully transferred between the inevitable succession of responsible incumbents. This responsibility for the functional operation of that system must have a continuous thread throughout the system life cycle.

Software Management Interactions

We have shown previously that there are really five distinct types of Army management involved in the complete life cycle of software that is a component part of any automated system. The implications of having five types of management we found to be worth investigating in some detail.

Our model illustrates that one of the most difficult problems for a program manager during the Full-Scale Development phase of a major system is that he must respond to the continuous management oversight of the functional or line manger and his changing requirements. Program managers would always recommend, if they had the choice, 'that the user or functional manager stay out of their hair during the development phase.

In addition to the curse of the user and his changing requirements, the program manager also has a problem if new technology is involved in that the technical manager will want to insure that any innovations developed by his scientific and engineering people are correctly incorporated into the new system.

But that's not all. The program manager has yet another individual to answer to, the logistics manager who is looking for better and more complete documentation, well defined system interfaces, modularized architecture, and evidence of the use of modern programming practices or structured techniques so that the system will be easy to maintain.

In spite of the problems these overseers may impose upon the program manager from time to time, he must incorporate their "parochial views" into his program plans. The fifth set of managers, corporate, are responsible for insuring that this is done in such a way to best serve the interests of the overall organization, be it the Army, Department of Defense or the Country.

In short, we feel that this conceptually explains both the necessity for five interacting types of software management and the difficulty program managers have in coping with this situation during the software life cycle.

Qualifications and Exceptions

There are, of course, qualifications and exceptions to the model presented in Figure 1. Some of the more important ones that should be considered are summarized below.

Degree of Structuredness. The degree of risk or novelty in a software development project or its "degree of structuredness" must be considered when using the model. Those projects which are very similar to others that have been done before, or those involving modifications to major systems generally do not require sufficient numbers of scientists and engineers to be employed to warrant the special aspects of technical management. On the other hand, a major project such as SAFEGUARD or APOLLO would obviously require many technical managers to handle the large numbers of scientists and engineers working on the system in their attempts to advance technology state of the art.

Project Size. Much time has been devoted in this and many other conferences to describing differences in life cycle management based upon problems of scale. For example, a short term, one-man project to design and develop applications software to solve an ad hoc engineering problem may have an entire life cycle as short as a few months. Clearly, the five varieties of management described above would not all be used. However, major defense systems that must undergo the Defense System Acquisition Review Council (DSARC) process normally involve all five types of management and have life cycles as long as 20 to 30 years.

It is sufficient to say that there is a drastic difference between a one-year development for a small system versus the 30-year life cycle of a major defense system.

Reliability Requirements. Software reliability for systems such as a management information system that operates occasionally using only historical information is clearly not as critical as for real time, operational systems involving missile guidance, nuclear safety or life support. In general, our group feels that, as system software reliability requirements increase, the need for more of the five types of management involvement also increases, thus lengthening the overall software management life cycle.

Other qualifications can, of course, be added to those described above. However, during the course of our deliberations we could not think of any qualification or exception that would negate the life cycle model as we have described it. That is not to say that one or more could not be uncovered through a more thorough analysis.

SLCM Milestones

Another major topical area discussed by the Dynamics Theory Group involved the milestones that delimit phases and subphases of the software management life cycle model described above.

Purpose and Nature of Milestones

The first question we asked ourselves was: Who are these milestones for? Our conclusion was that this question can have only one logical answer. Software life cycle management milestones should be designed for the direct use of the five types of management decision makers described above.

This led to a further conclusion that any milestone in a software life cycle management model must be a point of measurement at which information on the state of the life cycle process is collected for the sole purpose of serving as an input to help solve a specific management decision problem. It is significant that this is in concert with the principles of elementary decision theory which can be used to great advantage in improving all levels of software life cycle management. Thus, we emphasize that:

> Milestones should represent the termination of specific activities or tasks and also provide a measure of the degree of completeness (or quality) of those activities. Hence, they must be quantitative to be useful to management for judging the progress of a system through its life cycle. If they are not quantitative in nature, they simply cannot adequately support this necessary measurement function.

Milestone Taxonomies

A secondary question as to whether or not there are or can be various classifications or taxonomies of milestones we found not to be relevant to our other findings. Even though a manager who is interested in controlling a budget might use a different set of milestones than, say, a technical manager who is interested in keeping track of progress in the engineering development of a system, we feel that at some point in the management hierarchy there exists (or should exist) a single program manager who oversees the greater system that would be interested in all of the milestones regardless of how they are classified.

Iterative Process Milestones

One interesting conclusion reached by our group came after a discussion with respect to the meaning of milestones in an iterative environment. This environment, for example, is one in which a computer program keeps cycling back through its development phases perhaps because it has not passed an operational test. This could be a computer program that has been certified to be bug free and is in fact running satisfactorily, but has not yet completed the Fine Tuning subphase of the Deployment Phase.

The milestone that indicates termination of the Fine Tuning subphase must have associated with it a set of metrics which are used to judge the quality of that computer program. This quantitative data is used by a management decision maker to help him decide when that computer program is ready to proceed into the Modification subphase of the life cycle.

Implications of Milestones for Decision Making

Figure 2 illustrates a toy model of a decision tree which contains three possible states of nature that might occur at a milestone in a software life cycle. Branch A indicates that the software is of good quality, Branch B fair quality, and Branch C poor quality.

If the computer program is judged to be of good quality at that milestone, and the decision maker is viewed as sitting at Node 1, he might be faced with only a single alternative with respect to what the next directive to issue will be..."continue on."

If the computer program has not passed the criteria established for that milestone and is considered bad, the program manager can be viewed as sitting at Node 3. Now he is faced with multiple alternatives to decide upon. For example, should the program be killed? Should the program be allowed to proceed while that particular computer program is sent back for more work and so on?

Similarly, if we take the middle case at Node 2 where a computer program is determined to be marginal based upon a set of milestone quantitative criteria, there again would be several alternatives from which the decision maker must choose.

When we reflect upon this type of a model, we can see the striking resemblance to the program progress review briefings given to corporate managers by program managers with respect to various aspects of their programs. It is common practice to show categories such as budget, software, hardware, organization and schedule in red, yellow or green colors indicating "in trouble," "potential problems" or "no problems," respectively.

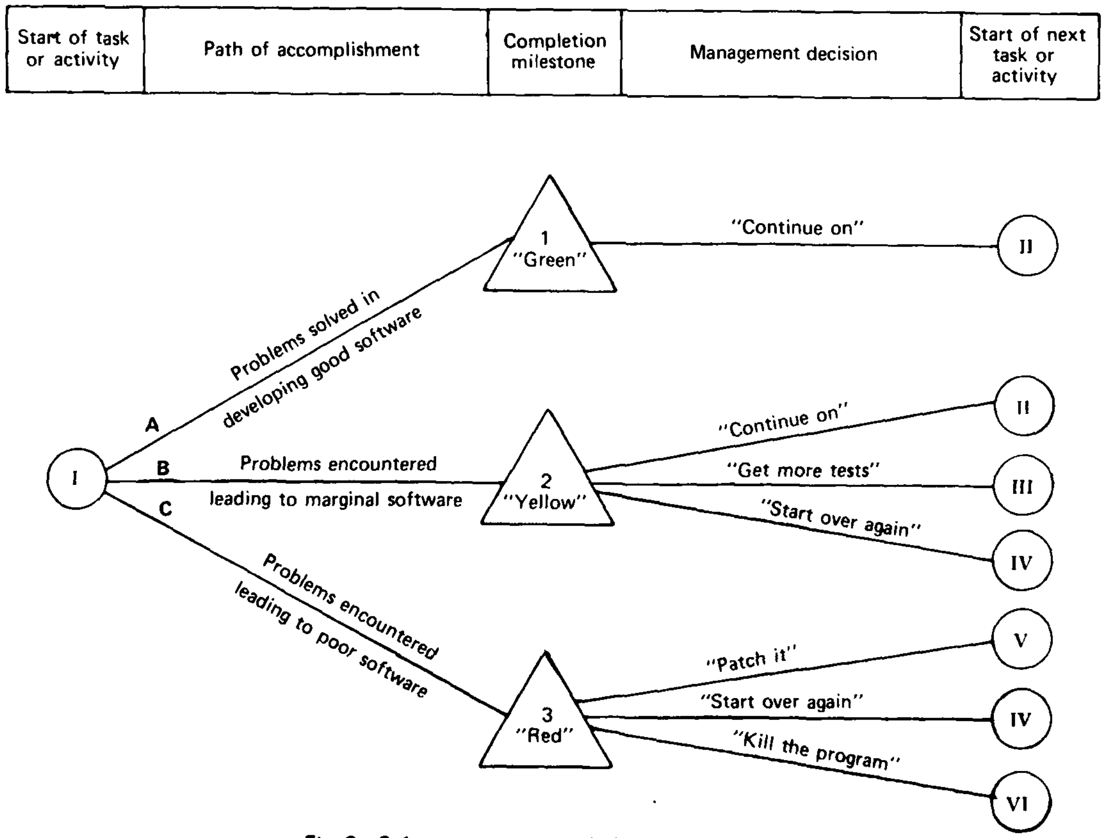


Fig. 2 Software management decision tree concept.

What this really means in terms of decisions, decision makers, and software life cycle management is simply this, a milestone is meaningless unless it has associated metrics. The metrics must be such that they provide meaningfull information to the manager at whatever level so that he can make appropriate decisions, that is, choose the best perceived alternative course of action based upon the information presented.

Milestone Metric Research

Now the basic question remains: How do we measure attributes of software such as quality or completeness at required milestones?

Some work is being done in this area but clearly not enough. An extremely important research direction for the Army to pursue is to develop a set of common measurable milestones for SLCM projects. Each milestone must have associated with it relatively easily measured attributes of the products delivered at the end of the activity it stands for.

Furthermore, we must all realize that progress of software through its life cycle is not measured by time, but by accomplishments. Thus, there must be some description of measurable criteria with which to determine physical evidence of progress. These milestone accomplishment critera must be complete enough and in sufficient detail that the resulting measurements will be sufficient to generate relevent management decisions.

Postscript

Federal ADP Reorganization Study

The workshop's featured speaker, Mr. E. Larry Dreeman, reported on the findings of the Federal ADP Reorganization Study National Security Team which he chaired. Mr. Dreeman's team was chartered to investigate ADP activities in the Department of Defense. This was a portion of one of 31 Federal government reorganization projects initiated by President Jimmy Carter. The overall project study objective was to investigate ADP in the Federal government and to recommend inprovements in the government's use of information technology; specifically, to improve delivery of government service, improve acquisition management and use of information technology and eliminate duplication and overlap.

Mr. Dreeman's presentation evoked a very active response from the audience as it appeared that many of the findings and, in particular, the recommendations were quite controversial. After the presentation, I had the opportunity to discuss several aspects of the study with Mr. Dreeman in more depth that led to his request for a formal input to the study. A summary of my personal response to this request is outlined below for the benefit of those who attended Mr. Dreeman's presentation.

Central Problem

My 18 years of involvement in the military computer resource management area has led me to conclude that:

> The primary cause of many of today's Department of Defense ADP management problems involves the obsolescence of the centralization of ADP management control policies.

This does not mean that I do not agree that there should be centralized policymaking and guidance to prevent chaos in this 8-10 billion dollar a year Department of Defense business area. The distinction I make is between policymaking and control.

Recommended Solution

I recommend that policymaking for technical issues continue to be centralized in the form of a Department of Defense ADP "focal point" to serve as an interface with other Agencies, the General Services Administration, the Office of Management and Budget and the Congress. In addition to the single interface function, the focal point will develop policies and guidance in the areas such as the following:

a. Standardized methodologies for the acquisition of automatic data processing equipment (ADPE).

b. Guidance with respect to standardization of computer programming languages for use within the Department of Defense.

c. Guidance and standardization of computer and peripheral interfaces within the Department of Defense.

d. Standardization of data elements and codes used within the ADP environment in the Department of Defense.

The problem with the centralization scheme as it exists today is that the authority for purchasing computers and associated software is retained at excessively high levels. This hinders effective equipment and software replacement programs designed to prevent the massive hardware obsolescense problem accurately reported by the National Security Team in its "Draft Report" (Reference 5).

Thus, I strongly recommend that:

The authority to purchase computers and software should be decentralized to the maximum extent possible subject to centralized technical standardization policies.

Furthermore, the purchasers of ADPE and commercially-available software in the DoD should use a "systems approach" similar to that used for the acquisition of weapon systems hardware and software using the Armed Services Procurement Regulation and as further defined in Air Force Regulation 800-14 (Reference 6).

This means that DoD functional managers should have more control over the type and quality of ADPE and software that is used within their areas of expertise and jurisdiction.

Relationship of ADP Management to ECS Management

We must recognize that the reason for the success of recent (I stress the word recent) defense system procurements involving embedded computer hardware and software lies not in the use of the "excluded" special-purpose, militarized computers but in the perspective of the program manager who is developing a system which is not an ADP system but rather an aircraft, tank, missile or spacecraft. This same embedded computer system philosophy can be profitably applied in the ADP environment.

For example, any DoD personnel manager should treat his ADP equipment, computer programs, supplies, people, computer data, and all other computer resources as but component parts of his "personnel system." The ADP manager should no longer be permitted to unilaterally dictate what specific types of equipment are necessary and allowable for the personnel manager to perform his function. A more effective role for the ADP "single manager" should consist of providing the personnel manager and other line and staff managers with technical advice and service as requested.

Arguments Pro and Con

Critics of the above "systems approach" to ADP management state that we would soon return to chaos in the ADP environment if we dismantle the current strict review process which proceeds from the lowest levels all the way up through the General Services Administration and, incidentally, takes nine months to a year and a half for approvals for equipment over $50,000.

I contend that realistic and well thought out policies developed by the centralized ADP policy officials can be enforced by existing Inspector General and auditor organizations within the Department of Defense. It is their job to make sure that all policies dictated from higher authority are, in fact, carried out at the lowest levels of all military organizations. Therefore, the technical policies, such as the use of specific higher order languages and standard data elements and codes can be easily checked for compliance during the normal visits to DoD installations by these enforcement agency representatives.

With respect to the argument that the costs could go out of control if this centralized authority is not retained at the highest levels, I contend that the budgetary process itself will hold down costs, probably to a greater degree than one can achieve with centralized management control. The reason for this is that the functional line or staff manager receives one slice of the budget pie as his total share for any given fiscal year. If it can be proven to him that a new piece of ADPE can reduce costs, improve organizational efficiency, increase functional system performance, or has any other purpose that is worth spending some of his limited monetary resources on, he will probably approve it. However, all functional managers within that commander's organization are also trying to get their own pieces of the same pie slice. This adversary situation with respect to the budgetary process is normal in all areas at the present time to the best of my knowledge, except for ADPE. I believe that if ADPE authority for purchase is decentralized to the lowest functional levels possible, the normal budgetary process will be much more effective in reducing overall ADP costs than does the present highly centralized management control system.

Finally, with respect to the potential problem of not having expert help if there is not a strong central technical organization that retains the authority to pass judgment on ADP purchases, I submit that a very effective organization already exists to help users with this function, the Federal Simulation Center (FEDSIM). This agency can be called in by any functional manager in the Department of Defense as a consultant to help them with the technical problems of performance evaluation, monitoring, equipment tradeoffs, make or buy decisions, source selection, competitive procurement, and so forth.

Postscript Summary

In summary, the three basic points I will be making to the National Security Team Chairman, Mr. Dreeman, are as follows:

a. A serious problem with ADP management in the Department of Defense is the obsolescense of Brooks Act implementation policies.

b. I recommend that ADPE technical issue policymaking be centralized to conform with the intent of the Congress for single focal point cognizance and reporting.

c. I recommend decentralizing the authority to purchase ADPE and software such that functional managers are given the authority to develop their ADP resources just as they develop all other resources at the present time.

Thus, I do not believe there are any compelling reasons for continued centralization for ADPE control at the highest levels of DoD management, and strongly request that the final National Security Team recommendations to President Carter be modified to incorporate these views.

In Conclusion

The Dynamics Theory Group lists the following conclusions and recommendations as those deserving further consideration by the Army in its attempt to improve software life cycle management:

1. Appropriate management methods that are applicable to different phases of the life cycle do not vary across different classes of software. However, the specific management method used can be and usually is different for different phases of the software life cycle.

2. We recommend that our proposed software life cycle management model shown in Figure 1 be used as a strawman baseline for a follow-on research effort by the Army to add conceptual detail to the individual life cycle subphases.

3. We feel that additional emphasis should be placed on the management activities in the Deployment Phase of the software life cycle, especially with respect to the problems of transfer of development management responsibility to the support management team. We recommend that the Army follow up on this point with further research to develop an effective system (software) deployment strategy.

4. We recommend that the Army focus a significant research effort on the critical problem of trying to determine the point when software "fine tuning" is sufficiently complete to be able to confidently transfer a system to the Support Phase of its life cycle.

5. The planned cost of software embedded in a system should terminate with the initial entry of the system into the System Modification subphase of the Support Phase of its life cycle.

6. The answers to the questions of whether or not ECS or ADP systems require different types of management practice or different types of life cycles are simply yes and no, respectively.

7. Milestones should represent the termination of specific activities or tasks and also provide a measure of the degree of completeness (or quality) of those activities. Hence, they must be quantitative to be useful to management for judging the progress of a system through its life cycle. If they are not quantitative in nature, they simply cannot adequately support this necessary measurement function.

8. An extremely important research direction for the Army to pursue is to develop a set of common measurable milestones for SLCM projects. Each milestone must have associated with it relatively easily measured attributes of the products delivered at the end of the activity it stands for.

Acknowledgment

As Chairman of the Dynamics Theory Group I wish to gratefully acknowledge the contributions of both the participants and the many attendees whose active participation provided the material contained in this report. In particular I thank the participants, namely, Thomas DeLutis, Melvin E. Dickover, Harvey Koch, J. David Naumann, Francis N. Parr, Leon Stucki and Robert Thibodeau for their individual contributions that helped make the workshop a most stimulating experience for all of us.

References

1. "Major System Acquisitions," DoD Directive 5000.1, January 18, 1977.

2. Manley, John H., "Embedded Computers - Software Cost Considerations," in AFIPS Conference Proceedings, Vol. 43, 1974 National Computer Conference, Montvale, N.J., AFIPS Press, 1974, pp. 343-347.

3. "Management of Computer Resources in Major Defense Systems," DoD Directive 5000.29, April 26, 1976.

4. "Brooks Bill," Public Law 89-306, 40 U.S.C. 759.

5. "National Security Team Report (DRAFT), Federal Data Processing Reorganization Study, President's Reorganization Project, 7 July 1978.

6. "Management of Computer Resources in Systems," Air Force Regulation 800-14, Vol. I, 12 Sept 75, and "Acquisition and Support Procedures for Computer Resources in Systems," Air Force Regulation 800-14, Vol. II, 26 Sept 75, Department of the Air Force, Washington, D.C.

II. LIFE CYCLE MANAGEMENT METHODOLOGY DYNAMICS PRACTICE

Chairman: Dr. Raymond W. Wolverton
TRW Defense & Space Systems

PANELISTS

George J. Schick	Harold Stone
Barbara C. Stewart	Ivan Jaszlics

Gerald M. Weinberg

SOFTWARE LIFE CYCLE MANAGEMENT - DYNAMICS PRACTICE

Summarized by
R. W. Wolverton

TRW

INTRODUCTION

Seven position papers are given here that reflect the practice and experience of each participant in the dynamics of software life cycle management. On the basis of his past contributions to the field, each participant was invited by the U.S. Army Institute for Research in Management Information and Computer Science (AIRMICS). Along with the position papers, the participants dealt extemporaneously with a list of questions offered by the AIRMICS technical chairman, Dr. Victor Basili. Three of the questions were selected for group discussion as having the most interest at this time and the greatest potential leverage in reducing cost and risk for future AIRMICS projects. The findings in these three areas, management dynamics, software tools, and life cycle maintenance, are summarized here. The participants for this session of AIRMICS 78 are:

George J. Schick, University of Southern California

Barbara C. Stewart, Honeywell Systems

Harold Stone, University of Massachusetts

Ivan J. Jaszlics, Martin Marietta, Denver

Gerald Weinber, Ethnotech, Inc.

Ray W. Wolverton, TRW Systems Group

Kenneth Kolence, Institute for Software Engineering

In addition, all attendees (approximately 50) participated in generating answers to the three questions chosen by the group (i.e., participants *plus attendees*).

OVERVIEW

R. C. McHenry and J. A. Rand of IBM contributed a position paper, although circumstances prevented its oral presentation. They believe that the very nature of top-down development allows it to become a powerful tool and technique for system integration, thereby leading to earlier and more complete system readiness than would otherwise be possible. Their key point is that by incremental development and testing, a new discipline is possible (termed transition management) that - properly implemented - does not require added development time for testing. This is an extremely powerful hypothesis for future AIRMICS study.

G. J. Schick and C. Lin of USC show quantitative techniques for determining the manager's preference for prior distributions that are needed in software reliability models using Bayesian probability theory. Predictive reliability models assist the manager is estimating the number of errors indigenous to the software system under development and the amount of time required to reduce the indigenous error population to an acceptably low level. Their solution lies in an automated question and answer dialog with the practitioner to find his level of indifference to alternatives that imply statistical fractiles. In this way mathematically tractable estimates of error-reduction can be made that incorporates the practitioner's software experience and intuition.

One of the stated assumptions is that software development is not a branch of mathematics but rather a special form of communication, person to person and person to machine. B. C. Stewart of Honeywell Systems offers a new discipline for alleviating the intrinsic difficulties of communication, particularly early in the design process, that combines an analysis model and analytical procedures. Her methodology has the benefits of assuring that both design goals and organizational goals are met, providing a means to evaluate the cost effectiveness of the organization's design methodology; and establishing a measn by which differing design methodologies can be quantitatively compared.

Harold Stone of the University of Massachusetts and Aaron Coleman of the U.S. Army (CORADCOM) report on their hardware/software life-cycle model that measures the cost of standardizing the computer instruction set together with the support tools for the military computer family (MCF). Their results show that the GYK-41 (PDP-11), out of the set of four semifinalists in the MCF study, ranks as the best choice for the MCF under their criteria of comparison. They use a 22-year interval for acquisition and deployment of candidate MCF computers and support software - 1980, 1985, and 1990 - with each lot deployed for 10 years. Their model is successful in identifying the critical cost-drivers and in estimating their relative importance, although it is not intended to predict dollar costs with accuracy. A crucial factor in the life-cycle cost analysis is that the greater the value of the software tool base, the lower the cost per line of applications code. Their model estimates the tool value as a function of time to reach time varying estimates of productivity. Their analysis shows that the GYK-41 (PDP-11) offers a cost savings of $1.5 billion over the next lowest contender.

C. E. Valez and Ivan J. Jaszlics of Martin Marietta, Denver, present a position paper on useful evaluation tools in the design process. Their hypothesis, supported by initial experience, is that design languages are emerging for identfying requirements, design components, and design specifications on the basis of which coding can commence. They believe that several design languages apply at different levels of the design process. One of the main purposes of a given design language is to provide the human the capacity for interaction beyond the first available solution to the best solution for his requirements. Two often neglected phases in a design language approach are included in their potential solution: the definition of the man-machine interface and computer resource requirements. They believe that an integrated software design concept is essential to comprehensive definition of the system development interfaces. The importance of program design languages for the upcoming generations of software cannot be over-emphasized.

Gerald Weinberg of Ethnotech, Inc., expresses his views on why the expected gains from programming tools have been slow in arriving, and often disappointing when they do arrive. He believes that the problem lies in the failure to understand the processes by which new technology is introduced. The role of training has been left by default to computers, under the assumption they are better or cheaper than human teachers. His solution is to provide an overall climate for professional learning in which both the computer has a role and the human teacher have a role. Probably all the tools needed to solve the elaphant's share of the software development problems have already been created, at the cost of perhaps a billion dollars a year, a micro-organismic sum of money has been spent on training people to use those tools. This means the practitioner does not use the tools accessible to him. We have spent billions for "tools", but not even pennies on understanding what is needed to create the professional technical leaders who will use them. The answer he advocates is to take computer training out of the realm of computers and put it in the brains of people.

R. W. Wolverton and B. W. Boehm summarize the more important lessons learned in developing a cost model for TRW. The key issues are first a need to develop agreed-upon criteria for the value of a software cost model, second a need to evaluate existing and future models with respect to these criteria, and third a need to emphasize constructive models that relate their cost estimates to software phenomenology and project dynamics. A potential solution to these needs is given by nine criteria defining the goodness of a software cost model. With respect to the emphasis on dynamics of the AIRMICS workshop, the structural form of the cost model can be used at RFP time (e.g., it does not ask for input data available only after design) and at all subsequent development cycle reviews. As more and more verifiable input data become available they replace the estimates, unknown data estimates are made current, and the manager can see the inception-to-date information on the estimated cost to complete and time to complete. Each cost element can be traced to a work unit in the work breakdown structures (WBS), and the manager can readily spot the trouble areas through management by exception techniques. A tie-in to the WBS is essential to show what is (and is not) included in the resulting cost estimate.

Kenneth Kolence, president of the Institute for Software Engineering, stands by his position that the field has advanced to the point where we now have a discipline of software physics. What is needed to tap this resource is an understanding of what this means relative to work performed, the capacity to take on new work, and the use of the metrics now observable. One then organizes the work of software design around a forecast for the use of facilities. The problem regarding software acquisition scheduled for the mid 1980's is to define in 1978 what data is really needed, collect it, analyze it by the laws of software physics, and incorporate it into an action plan.

MANAGEMENT DYNAMICS

Any person who influences the irreversible use of resources is in truth a manager whether his title is that or not. Dynamics deals with forces and their relation primarily to motion or time behavior but sometimes also to the equilibrium of the acts of management. The word also implies patterns of change of growth. Therefore, management dynamics refers to the forces inherent in the process of leadership and their interplay as a function of intensity and frequency.

Startup Conditions

Software management has the same major components of organization and enactment as the successful management of any other complex human endeavor, with two significant observed exceptions. First, managers do not generally recognize that the software development process has more degrees of freedom than another project of equilivent dollar value. This unique quality accentuates the recurrent people-oriented difficulties associated with span of control through not knowing precisely what to control, by what criteria, at what timely milestone events.

Senior managers are now in decision-making roles by a traditional reward system based on earlier career successes, where success is defined by the manager's management. Often, the recognition is based on earlier projects for which the software component was smaller and the criteria of success not clearly focused. These early projects were often managed through intuitive, and undescribable, methodologies. Inevitably, this situation is accompanied by schedule slippage, cost overrun, and low-quality field software.

To further aggravate this condition, these (now) older in situ managers are overtaken by events in the form of accelerating advances in computer technology and still more degrees of freedom, increasingly difficult management decisions (shifting from, say, two-valued deterministic choices to multi-valued probabilistic choices), and the conflict created by the need for "detachment" for the sake of overall visibility on the one hand, and the need for "involvement" for the sake of in-depth understanding on the other hand. If the top-level manager does not understand the complexity of the dynamics of the software development process he is in difficulty. What a manager does not understand he cannot manage. His recourse, often adopted unconsciously, is to move from active management to reactive administration. He is now driven by events, and the otherwise manageable project becomes unmanageable.

Paradoxically, too much "understanding" of the problem to be solved gives rise to too many good ideas for its solution. This can lead to expensive gold-plating at one exteme or paralysis and delay at the other. Moreover, always underlying the management of software is that software tends to be invisible unless made visible.

Second, managers do not generally recognize that a software project is of the same nature as a comparably sized non-software project. Software implementation needs a capital base for establishing an overall professional working climate, i.e., modern computing facilities, compilers, operating systems, effective support tools, trained people, and a support group including configuration and data management specialists. The work is done by human beings, not machines. Some managers do not acknowledge that software is, in fact, manageable, and this leads to self-fulfilling prophecy that it is not. Software is a thing, a product, an asset, it is as real as hardware and can be managed.

Ideal Management Circumstances

The ideal ingredients of software management can be identified unambiguously. However, it appears that some things are so obvious as to be overlooked or not applied when they are easily within the manager's grasp. Management should accept and define the problem and set team goals. He should not accept fuzzy or ill-defined project requirements; other non-software technical disciplines would reject many jobs readily accepted by software managers. The most crucial step is for the manager to properly subdivide the problem into manageable-sized packages of work and assign clear lines of authority and commensurate responsibility.

He must assign tasks to individuals (e.g., sub-project managers) and set individual goals. One way to do this is to define subordinate objectives that, in turn, support senior objectives. The manager assigns a measurable task to an individual for completion by a certain date at a cost not to exceed so many dollars. Then the manager can constructively monitor and assist the work objectives by periodic reviews organized to compare outputs against objectives, including cost and schedule predictions versus actuals. A work breakdown structure is a proven technique for helping to avoid ambiguity. He has existing tools to aid him in doing an effective job.

Management Aphorisms

Here is a collection of aphorisms put forth by this group during the AIRMICS 78 workshop.

a) Ensure that common standards apply to all parts of a project. Ensure that the interfaces between modules are managed at a high enough level for the consequences of any change to be appreciated. No one can enforce an order that the consequences of a change be appreciated.

b) Simple projects can be managed by a traditional "scalar chain" line-staff relationship. Complex projects, which are more the norm, require staffing from different disciplines. A matrix organization is required. Then conflicts ensue between line and project; this is called divine discontent.

c) Within a matrix organization the emphasis in technology will shift over the useful life of the software. The manager's management should have a written plan to change the organization to suit the demands of the software evolution. If the plan is not written down it does not exist.

d) A manager should have a technical background and explicit training in management skills. The manager should be relieved of the requirement that he is the most technically competent. He should have the intrinsic ability to motivate and develop loyalty. This circumstance now meets the criteria of Murphy's Law.

e) The ideal manager has superb management training, but nobody notices.

f) Everything good happens early. Unmanaged projects do not, unfortunately, have a high infant mortality rate. To bring an unmanaged project under control requires changing of the project manager, revising the project plan, developing a new schedule and budget, and revising objectives. Otherwise, there is nothing to it.

g) If you are a good leader who talks little, they will say, when your work is done and your aim fulfilled, "We did this ourselves" (Lao-Tse). This view does not succeed east of Los Angeles.

h) Few programmers become major officers of the company. It is probably true that programmers have a less clearly defined career path than most professionals.

i) To sell an idea to management, make sure management thinks of it first. There is nothing that cannot be accomplished if one doesn't care who gets the credit. The fact is that everybody does care who gets the credit.

j) No project will succeed if the energy is directed toward placing blame. One can find out if a project is beginning to be in trouble by asking the secretary who in the project is building a white file.

k) Hardware is built from documentation. Software is built and then documented. This documentation is often for an earlier version than "as built." Matching the documentation to the software as delivered is a management goal, frequently unrealized.

l) Decisions regarding hardware design and implementation are nearly irrevocable, whereas, the software manager can naively operate from the false premise that he can correct faulty decisions at a much later stage in the development process. He gets to have this attitude for one project only.

m) Some human behaviorists who think about "management success" are grounded in the view of Machiavelianism, especially the view that politics is amoral and that any means however unscrupulous is justifiable in achieving political power. Many successful managers become politicians and vice versa.

SOFTWARE TOOLS

A tool is something necessary in the practice of a vocation or profession, as a scholar's books are his tools. This section gives an overview and minimum detail on the wide-ranging concerns of this group on software tools.

Table I shows a few up to date software tools for typical development cycle stages. Literally hundreds more exist. If one views maintenance, especially enhancement maintenance, as a series of mini-development cycles, one can apply essentially the same tools to the sequence of planned enhancements during maintenance. Under these conditions a given tool may have a useful lifetime of 10 or more years. Although this group did not consider the quantitive aspects, say, of the breakeven cost for tool development, a plausible case will be made for the cost effectiveness of soundly conceived (and transportable) tools based on group experience.

Group Observations About Existing Tools

The key goal regarding the future of management information tools is increased visibility by the project manager at any time. This goal implies walking to a local terminal and getting project actual versus planned expenditures, estimates of time and cost to complete, and other statusing indicators especially management by exception indicators, for example, the cost from inception to date for a particular module exceeding a predetermined threshold such as 10 percent, within a certain period.

Four steps are involved in reaching this goal:

a) Clearly define the management information process in an organization.

b) State requirements for tools.

c) Implement tools starting with the areas with the highest payoffs.

d) Audit existing tools for current effectiveness. Identify candidates for replacements.

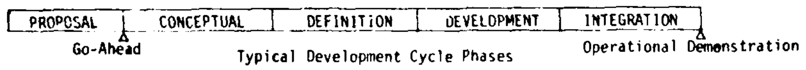
One premise is that the managment process is driven by the product structure. Management tools must be unified with product description tools. The tools must be interactive in the sense that task networks can be readily modified as the design developes. The task partitioning problem is drīven by the design partitioning problem and the dynamics of circumstances over time.

Project aggregates (i.e., man loading versus time and modules complete versus time) should be made accessible on a periodic or demand time basis. To meet this goal the actual progress must be compared against the planned progress. The ability to estimate needed resources from a skeleton design is implied.

Product design tools may be categorized as existing tools or advanced. Existing or available tools customarily include assemblers, linkers, program preparation aids (editors, etc.) debug assist tools (path tracers, etc.) and high level language manipulators (compilers, etc.). Advanced tools are defined to cover very high level languages (automatic-program generators, etc.), products to assit in creating and debugging executive and real-time programs, products to assist to performance analysis, and products to assist in verifying distributed data processing configuration.

Table I. Existing Software Tools

Tool Name	Function	Life Cycle Phase	Language and Computer	Where
Software Cost Estimating Program	Estimates effort for development cycle in man-months, by phase	Proposal and major milestones	Fortran CDC 66XX	TRW
Software Requirements Engineering Program	Analyzes requirements by relational data base	Conceptual	Pascal CDC 7600	TRW
Manuscript Preparation System	Text is entered to a computer file by remote terminal and edited	Conceptual and as needed	Compass CDC CYBER 74 CDC 6000 Series	TRW
Program Design Language	Design is written in structured English by six control constructs	Definition	Fortran 360/370, 1108, 6XXX, DEC-10, SEL 932, PDP 10/20/11	Caine, Farber and Gordon, Inc.
High-Powered Accounting Resources Program	Provides graphic display of scheduling and resource information	Definition	Fortran 6XXX, Calcomp	TRW
Performance and Configuration Analysis Model	Represents systems by event logic tree's before any code exists	Definition and development	Fortran CDC 6X00, 7X00	TRW
Software Design and Verification System	Supports design, code, test and maintenance of DAIS mission software	Development (simulation tool)	Jovial J73, Cobol, Assy, Fortran DEC 10	TRW



Typical Development Cycle Phases

Why People Do Not Use Tools

This group believes that the topmost issues is the need to invest time to educate and train the practitioner. As viewed by the manager, the payoff for investment of time is not clear and not presently well documen'ed or understood.

Some existing tools work poorly and bias managers (and programmers) against future use of any new tools. Programmers do not willingly learn about the existence of new tools accessible to them. Progressive change is difficult to instill. Standardized abstracts and ascession lists would be helpful here, especially in taking away the unknown amount of research into the structure of the tool and its utility from questionable sources. Tools are not easily transported; what worked well for one project may be commercially impossible for another project because the computer and configuration or language is significantly different.

In sum, people do not use tools because:

a) They do not see a direct benefit to them.

b) They do not understand the specific tool and perceive a high risk of failure, low chance of success, or poor initial tool behavior that could be blamed on them.

c) Management has coerced the project as a whole into using a specific tool, despite inadequate training and planning for its introduction. People wait for somebody else to be first.

d) They are pressed to meet a difficult schedule and have no time to experiment with a new tool or the accompanying new techniques.

e) They perceive that the proposed tool does not work at all in their particular environment. On the other hand, they may not know the tool exists.

In contrast, people do use tools because they see a direct benefit to them, management encourages them in various ways, they see a good chance of success (and no alternative without it), it is new and exciting, there is good introduction and training strategy, they have slack time in which to experiment, they are rewarded by management measures for its use, and the tool is a part of a package they use anyway.

Intrinsic Advantage of Tools

In the sense used here, tools should encourage a standard approach to solving a recurrent problem. The form of the output should be such that it directly meets the informational requirements of immediate and successive activities. For example, a project management tool should generate reports that directly compare the actual completion status with the planned status without the inconvenience of working among multiple reports.

Tools can and should encourage improved human performance by putting off tedious or repetitive actions that can be done by an algorthm into a computer. The person is then free to look for patterns, trends, and relationships, analyze results, and bring to bear his own creativity.

Tools can and should encourage a professional attitude toward work. The individual can more easily experiment and do creative tasks. He is producing openness and visibility into an otherwise invisible process. He can improve product quality by systematically reducing indigenous errors and know that he has done so. In turn, the manager sees an effective allocation of tasks between humans and machines according to quality and then efficiency if quality has been assured. More consistent results means more manageable results.

Tool Selection Criteria

The proposed tool should automate any repetitive part of the programming or design task, increase productivity, accuracy, and improve morale.

The proposed tool should automate the extraction and correlation of various data files containing management or technical information about the program.

The proposed tool should assist in formalizing software development procedures to assure a consistent management approach and development methodology. A tool should enable people to operate at a higher level of competence. Some things only a tool can do and do well, e.g., indicate which branches of logic have and have not been exercised for a given level of testing and a selected data state vector.

The proposed tool should:

a) Meet all known requirements, have capacity for growth, accommodate a reasonably wide application of use, and have a long expected lifetime.

b) Be maintainable, transferable (within technical limits of the intended environment), and adaptable for training.

c) Fit well with other tools already in use or planned. It should compare favorably with the capability of the tools it will become a part of.

d) Compensate for a deficiency in resources or organizational layout. It should meet cost/benefit criteria established by group standards.

e) Deal fairly with human factors, for instance, by converting from arbitrary internal units to external engineering units understandable by a person.

LIFE CYCLE MAINTENANCE

By commonly accepted practice, the software life cycle consists of the development phase and the maintenance phase taken collectively. Contract developers have been known to talk about the "life cycle" when they really mean the "development cycle." This clarification must be made at once. A typical distribution of resources for a large-scale software project might be 10C people for two years in the development phase and 35 people for eight years in the maintenance phase.

Basic Maintenance Circumstances

In this hypothetical case, the development phase exists from contract go-ahead to operational demonstration and government acceptance by DD Form 250. The maintenance phase, by definition, is then the interval when everything that is not development happens to the software package. Of the 480 man-years hypothesized, 200 man-years is expended in development and 280 man-years in maintanance. Thus about 40 percent of the life cycle resources are given over to the building (i.e., development) and about 60 percent to keeping the software package in the existing state of readiness, efficiency, or validity (i.e., maintenance).

With the majority of the out-of-pocket expense to the government going into something we all call maintenance (60 percent versus 40 percent), unquestionably the most important factors in maintenance need to be examined closely. Unfortunately, although maintenace is the most visible and costly phase it is the least documented and understood. In some areas of specialization (GTE-AEL), they have a defined life cycle of about twenty years (i.e., the life of a telephone switch). Probably in this example, good records of reliability and cost are available but the ability to generalize from the specific is not productive for, say, a command and control application.

One reason the industry does not have good answers to strategic-planning issues in maintenance is that the government writes a contract for development and then writes a different (annual) contract for maintenance, often under a level of effort (LOE). All the thinking goes into development because the LOE type of maintenance is managed by a policy of sequence and priority. The maintenance contractor, who may or may not be the developer, accepts any and all work required, provided that each task is listed (sequenced) by the customer's priority needs. Not a great deal of strategic planning is needed in this case, since the manager is reacting to his customer's needs on a day to day plan. Usually any task appearing on the lower 80 percent of the list is never completed (Pareto's Law).

However, there are some things that can be said and some intelligent steps the contract developer and the government can take to deal with this circumstance. The group considered four key concerns with varying levels of detail: enhancement maintenance, introduction of a new system or repair of an old system, design tradeoffs for maintainability, and effects of maintenance of development.

Enhancement Maintenance

One way to deal with the question of predicting and measuring the effective life span for a software system is to adopt the point of view that a software system will be around forever. Let us define "forever" as 20 years. After the original package is delivered to the operational user, the new point of view is that the original version will continuously evolve under a mini-development cycle concept. Then, everything the contractor does in development is just repeated in the enhancement maintenance circumstance.

Feasibility studies, tools, management progress monitoring, test and acceptance are carried out just as in the original development. Except now it is harder. Less core space is available and more skill is required to shoe-horn in the add-ons. Arbitrary style of programming may have been replaced by structured programming or by a program design language so the interface with existing software is more difficult. Regression testing requires more skill, perhaps more sophisticated tools, and a test rationale (perhaps with analytic models) must be available for judging how far back to go with the test procedures to verify mission readiness given that changes were made to the previsously accepted code.

Two computers may be required, one to use online to keep the operational program going and another to develop the product improvement in consideration of debugging, system testing, and operational readiness demonstration. The main thrust of the discussion is that enhancement maintenance can be thought of an extension of the development process. The release of product enhancements is usually by block release, i.e., work can go on continuously but new software is introduced incrementally.

Introduction of a New System (When to Redo an Old System

If we try to deal directly with the question of effective life span, we are faced with a dilemma. We believe the problem cannot be solved in a good or explicit way. The question of technological obsolescence is not independent of economic considerations such as life time ownership, variable (and unknown) user requirements, rapid new technology advances, and other practical considerations. Here, for example, a contract developer could not rationally allow a budget percentage to cover maintenance problems of all kinds. He would not be cost competitive in his initial proposal. Experience shows that if an equally qualified builder is more than

10 percent higher in cost than the lowest bidder he will most likely lose. He cannot realistically support an argument that in the long run his higher bid cost will prove to be lower when considered over the entire useful life time of 10 or 20 years.

After a certain point of growth, even with modern virtual memory machines, the system will have to be redone. Or, more easily argued is that the software has to be redone because the user wants to put an existing program (with some enhancements) into a completely new computer configuration: new language, new compiler, new data storage and transfer methods, and revised protocols for real-time interrupts. The question, however, is why put an existing and mission ready software package into a new computer configuration? In a more general sense, why are the perceived attributes of the existing software package not equal to the job at hand?

There are no inherent properties that can be used to measure (predict) the life span of a system. Given that we can all agree that the effective lifespan in finite, the life span can be observed to be over when:

a) The unit cost/transaction exceeds the projected unit cost of a new system by a sufficient margin. This estimated cost should include re-run costs due to low reliability.

b) The function that the system serves becomes obsolete within the sponsoring organization. For example, a plant may be closed and a software system (e.g., process control) designed to serve it no longer serves any purpose.

c) Defense measures require that a heretofore benign ground system must be transportable and the computer hardware able to survive a particular nuclear exposure.

In short, it is time to redo the old system when it is judged incrementally cost-effective to recreate the system than to enhance it further. This can occur because radical changes in the functional capability are required, or because the introduction of new hardware makes possible such a radical change. It can also occur when the effort required to maintain and to enhance the existing system begins to grow so rapidly that it is cheaper to redesign it. If a suitable measure of complexity can be found, a plot of this measure against system age is a valuable indicator of when the system is approaching a state in which maintenance or enhancement is no longer practical. Some evidence suggests that there is an upper limit to the size of a change which may be made to a system at one time, and that the original structure of the system is a severe constraint on the nature of the functional changes which can be accommodated without initiating a complete redesign.

Design Tradeoffs for Maintainability

Too many factors are involved in designing for maintainability to consider the issue thoroughly. However, some isolated findings offer promise for future AIRMICS studies:

a) Greater effort in the beginning (definition and design) by stressing ease of understanding, modularity, and ease of use (human engineering). This will be a commercially impossible goal unless some reward system is built into the government procurement policy (i.e., an incentive for doing more costly work).

b) Emphasis on software quality assurance and use of all available techniques to ensure correctness early in the software life cycle. The more promising reliability models should begin to be introduced into development test and beyond.

c) Use of machine assists to detect programmer errors.

d) Use of library maintenance and other tools to assist in correct changes to a program.

e) Greater emphasis on adaptability. Experience shows that large systems suffer continual change in the first 12 months and that after 2 to 5 years typically very little of the original code is left in the system.

Effects of Maintenance on Development

Maintenance considerations play a role in all phases of development. At each stage the product of that stage is examined (at every design review, code walk-through, unit/string testing etc.). The results are reviewed and a decision made as to whether the stage should be reiterated, realizing that deficiencies accepted at an earlier phase will result in maintenance difficulties later.

Perhaps the only workable approach to reducing the cost of maintenance is for the government to adopt a procurement policy in which the government explicitly acknowledges maintenance as part of the life cycle by the kind of procurement contract applied in the first place. Good procedures now exist for orderly change procedures to be built into the contract. However, the implication usually is that the sponsor and the contract developer are in adversary roles. Constraint will be needed by the government program office in keeping their requirement stable, which in turn keeps the design and test activity stable and matched to its necessary and sufficient mission demands, no more and no less.

MODIFIED DELPHI METHODOLOGY

In this group we used an each-to-all Delphi method in working toward group analysis of a common problem. A position was given orally by each participant, and these appear as a written paper in the next section. Where clarification was needed, questions were asked about a given position. Basically, more information was being transferred than there was time to dig into all of it. Questions were posed and the group narrowed them down to a somewhat manageable list. There were many more questions than answers, and a fact probably worthy of notice. Research is being applied to many problem areas, but the more problems that are solved the more questions are asked. Man's reach shall always exceed his grasp.

One small facet of the interesting group dynamics that occurs when many competent people are in the same room grappling with the same questions is given in the next two worksheets:

a) Pre-discussion self-rating sheet that is concerned with the participant's image of himself with respect to his peer group before getting into substantive issues. In this case the results include this session, and John Manley's Session I. The mean ($\bar{x}$) and variance (s) is indicated graphically on each question.

b) Post-discussion evaluation sheet that is concerned with the participant's evaluation of the results of two days of group endeavor to answer questions of interest to the administration of the AIRMICS 78 workshop. As shown almost unanimously the participants see the major problems as people-oriented and hardly at all machine-oriented. Such a conclusion is probably not very surprising. However, the worksheets may help the reader to understand some of the comments rising from this session. Every statement made cannot help but reflect the cultural attitude, subjective bias, and knowledge of each participant.

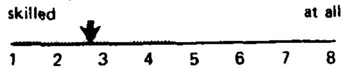
SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP
Atlanta – 21 - 22 August 1978

Code Number ____________

I. Pre-Discussion Self-Rating Sheet

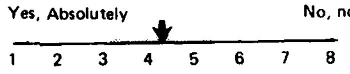
1. As an SLCM participant, my skills in software life cycle management would put me about here, relative to others.

Very highly skilled — No skill at all



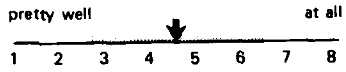
2. I think my ideas are in basic agreement with the rest of the participants.

Yes, Absolutely — No, not at all



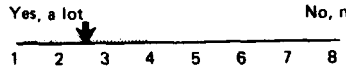
3. I know most of the people in the SLCM workshop very well.

Yes, pretty well — No, none at all



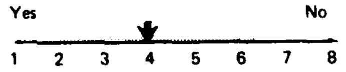
4. I have some definite ideas about what the goals of the AIRMICS SLCM workshop are and should be.

Yes, a lot — No, none



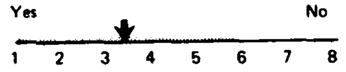
5. I have been in software life cycle management for longer than most of the other people here.

Yes — No



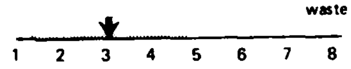
6. I have a lot of experience in SLCM practice outside of a university environment.

Yes — No



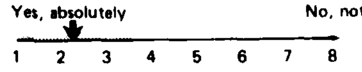
7. I am anticipating that the SLCM conference is going to be a good thing for goal-setting.

Yes, I think it will be — No, I think it may be a waste of time



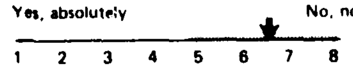
8. My approach to problem solving for this conference is best described as "people oriented."

Yes, absolutely — No, not at all



9. My approach to problem solving for this conference is best described as "machine oriented."

Yes, absolutely — No, not at all



SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP
Atlanta – 21 - 22 August 1978

Code Number ___ ____

II. Post-Discussion Evaluation Sheet

1.	I feel satisfied with the results in general.	1 [✓ 2] 3 4 5 6 7	I'm not really happy with the results at all.
2.	I got some ideas from the feedback.	1 [✓ 2] 3 4 5 6 7	I didn't learn a thing from the feedback.
3.	In general, I agreed with the ideas in the feedback.	1 2 [✓ 3] 4 5 6 7	I disagreed with everything in the feedback.
4.	I could express my ideas OK this way.	1 [✓ 2] 3 4 5 6 7	I couldn't really express what I wanted to say.
5.	I feel as if I really wanted to talk to people.	1 [✓ 2] 3 4 5 6 7	I didn't feel the need to talk at all.
6.	I think people understood my reasons pretty well.	1 [✓ 2] 3 4 5 6 7	I have a feeling people didn't understand or think about my reasons.
7.	I think the SLCM conference structure could be operational in goal setting more generally.	1 2 [✓ 3] 4 5 6 7	I don't think this SLCM conference structure could be operational at all.
8.	I think it went too fast.	1 [✓ 2] 3 4 5 6 7	I think it went too slowly.
9.	I would be pleased to attend the next AIRMICS conference.	1 [✓ 2] 3 4 5 6 7	I would not wish to attend again.

III. LIFE CYCLE MANAGEMENT MEASUREMENT MODELS - PREDICTIVE

Chairman: Mr. Lawrence H. Putnam
Quantitative Software Management, Inc.

PANELISTS

Barry W. Boehm	Bev Littlewood
Amrit L. Goel	John D. Musa
Mier M. Lehman	Robert C. Tausworthe

Marvin V. Zelkowitz

LIFE-CYCLE MANAGEMENT MEASUREMENT MODELS: PREDICTIVE

SUMMARIZED BY

Lawrence H. Putnam

Quantitative Software Management, Inc.

Current life cycle models have been inadequate to predict cost, schedule, quality and reliability. Group III examined the problem from three perspectives: management issues, phenomenological behavior and reliability measurement and prediction.

Central to the thinking of all was the notion that models were needed that provided adequate accuracy, faithfulness to the process, simplicity of use, timliness, and addressed investment and management questions directly in management parameters -- time, cost, manpower, cash flow, rate of progress, effectiveness and reliability.

The fundamental issues identified are:

- Lack of standard definitions and metrics for the life cycle
  - activities
  - phases
  - milestones
- A detailed process model is needed
- A catalog of existing descriptive and predictive models is needed. The catalog should contain:
  - Description of model
  - Assumptions
  - Purpose
  - Capabilities (positive and negative)
- A careful evaluation of existing models is needed (This should be done interatively with the creator to be sure that important characteristics and nuances are not omitted in summarization).
- A good life cycle model should possess these characteristics:
  - Consider all activities and phases
  - Relate management parameters to management responsibilities

	Plan	Control	Evaluate
Function	. .	. .	. .
Cost	. .	. .	. .
Time	. .	. .	. .

  - Be adaptive to actual project data and requirements changes (i.e., must be time-varying or dynamic).
  - Provide engineering accuracy (and uncertainty measures until it is safe to ignore them because of standards conventions (e.g., building codes, electrical codes, etc.) in cost, schedule and quality
  - Provide sensitivity profiles.
  - Be phenomenologically based.
  - Relate produce to resource comsumption (both statically and dynamically) and the technology being applied.
  - Be capable of future growth.
  - Be able to adequately treat known and future system types and development environments.

This group was composed of: Lawrence H. Putnam, of Quantitative Software Management, Inc., Chairman; Barry W. Boehm of TRW Defense and Space Systems Group; Amrit L. Goel, Syracuse University, Meir M. Lehman, Imperial College of Science & Technology/England; Bev Littlewood, City University/London, England; John D. Musa, Bell Telephone Laboratories; Leon G. Stucki, Boeing Computer Services, Inc.; Robert C. Tausworthe, Jet Propulsion Laboratory; Claude Walston, IBM Federal Systems Division; and Marvin V. Zelkowitz, University of Maryland.

The Group III people found it worthwhile to sub-divide themselves into three sub-work groups, and to devote their attention to the special areas of expertise in which they could deal with the subject matter in greater depth and address a smaller sub-set of the questions posed by the Army in a more comprehensive manner. Three subject areas were to be explored. (1) Reliability

Models. The people who worked on this sub-task were John Musa, Bev Littlewood, and Amrit Goel. (2) Life Cycle Models. The people concerned with this sub-task were M. Lehman, Claude Walston, Marvin V. Zelkowitz. (3) Management Issues and the Resource Control aspects that fit within the managerial framework. Barry Boehm, Bob Tauseworthe and Leon Stucki addressed these topics.

We will consider these in the order of Management Issues, Reliability Models and, finally, the Life Cycle Models.

MANAGEMENT ISSUES

The Management Issues sub-task group con-ed itself with the following questions: What are the major ingredients in the management of software? What makes it unique? What makes it different from hardware? How should the organizational structure relate to the problem to be solved in the different phases of development? To what extent should managers be technically trained? To what extent should technical personnel be managerially trained? Are there different classifications of software that require different methods of management (For example, embedded computers and software vs. non-embedded computers and software)? Are there predictable crises in the software life cycle, and if there are, what are the early indicators associated with these crises?

Embedded within these broader questions then is the overall set of fundamental questions which we hope will be answered. They relate to what needs to be done to improve the process. We might define these related actions under the broader heading of impirical studies--basically, what we need to know to understand the process better. What information should be collected about the process, the product, and their interaction and for what purpose? What kind of experiments should be performed? How can we capture and express the idea of program complexity? How can program managers be convinced to conduct experiments on their programs? What progress is being made on the transfer of learning from one project to another project within an organization and between organizations?

An integral part of management, of course, is resource planning and control. Within this framework are those things having to do with performance measures that will measure the actual progress of a project against some time base which we commonly called the milestones and in terms of rate of expenditure of the resources allocated to the project (which typically are: manpower, dollars, computer time). The ability to relate performance measures to the consumption of resources has been especially difficult in managing software products. Attempts have been made in terms of productivity. Productivity has been defined as total number of delivered source lines of code divided by the effort required to produce the code. Basically, people are unhappy with this definition in that it doesn't really relate to the rate of progress on the project. It is a difficult measure and in some sense may be counterintuitive to the common industrial interpretation of rate of production used in the context of the industrial production line.

The management sub-task group wrote these managerial concerns into a group of problems in which they identified the key factors, recommended an approach, and gave a prognosis with respect to possibilities for success and the time frame in which it might be possible to achieve the solution to the problem. The first problem identified by this group was problem: The inadequate accuracy of current models. This has resulted in frequent overruns. Parameters are often difficult to estimate, and the non-standard terms and metrics that are used in these various models complicate interpretation.

KEY FACTORS:

There are no standard metrics and terminology within the industry, or within major subdivisions of the government. There are inadequate empirical applications of the models (i.e., there is no practical application and subsequent feed back so that the models are self-improving).

RECOMMENDED APPROACH:

Establish standard definitions; establish refined data collection procedures; collect additional empirical feedback leading to refinements and tuning of models to make them better.

TIMING:

Reasonably near term (2-3 years).

● PROBLEM: Models need to be evaluated with respect to a set of management-oriented criteria.

KEY FACTORS:

Timeliness, updatability, definition, objectivity, detail, parsimony. The models should be extensible, contractible, tailorable. These should be a pragmatically understandable correspondence between criteria. The models should support sensitivity analyses. And the models should be adaptive, that is, they should respond to the project dynamics; what is actually happening should be fed in as it occurs and the model should adapt to that in terms of the future projection it makes for the next few time intervals.

RECOMMENDED APPROACH:

There is a need to extract meaningful management criteria from these Key Factors. Evaluations should be performed to establish a standard accepted set of terminology to develop new classes of models that will handle a broader range of phases and activities within the soft ware development and maintenance process.

TIMING:

Reasonably near term (2-3 years), leading to longer term pay-offs in the medium range (5-7 years) period.

● PROBLEM: Current models are not well related to the project status indicators.

KEY FACTORS:

The definition of status indicators, (for example, CDR, or Critical Design Review). Obsolute software standards (e.g. MS-1521 and MS-881). Inadequate detail (e.g. work breakdown structure, and lower level milestones).

RECOMMENDED APPROACH:

Define a more detailed life-cycle process model (include a greater number of lower milestones within the work breakdown structure). Relate global status indicators to the detailed process model. Update the relevant software standards. Relate the predictive models to the detailed process model.

TIMING:

Reasonably near term (2-3 years).

● PROBLEM: Current models are inadequate in relating productivity and reliability.

KEY FACTORS:

The terms are difficult to define. There are no standard definitions.

RECOMMENDED APPROACH:

Develop new models relating productivity with reliability; establish standard accepted definitions that adequately describe in a meaningful way productivity and reliability features that we want to see within the models.

TIMING:

Reasonably near term (2-3 years).

● PROBLEM: The current models do not adequately cover some key issues:

- maintenance, conversion, block updates
- the impact of new technology

KEY ISSUES:

- Understanding the underlying phenomenology of the software building process and how to use it in the model.
- the unknown domains of applicability of the models.

RECOMMENDED APPROACH:

- determination of areas of applicability for existing models (include underlying assumptions).
- develop additional models to cover the poorly developed issues and areas; this implies more detailed definitions and a greater data collection effort.

TIMING:

Reasonably near term (2-3 years). Some areas will require better data for significant pay offs and this will necessitate longer periods of time within the mid-range period (5-7 years).

● PROBLEM:

A lack of models for other areas of management purview

- other resources (e.g. *core requirements*), other situations (e.g. distributed networks, micro-computers)
- personnel career progression
- life cycle dynamics of software as a "Business Game" model similar to what is now done in the large business schools in which a complete business scenario can be played out and development outcomes determined depending on the input and the actions of the players.

KEY FACTORS

- Complete absence of models of this type
- Non-standard situations and organizational structures within various business entities and various government organizations.

- subjective versus objective decision making
- requirements for such models are not recognized

RECOMMENDED APPROACH:

- create awareness for the value of such models
- develop model goals, requirements, criteria, etc.
- develop adequate models
- train management to use the models in the decision making process
- sell management on the utility of using such models

TIMING:

Reasonably near term (3-5 years). Prospect for success is good in modeling quantitative measures (e.g., life cycle dynamics, core).

RELIABILITY

The reliability subtask group address the general set of questions concerned with models but specifically directed their responses toward reliability-oriented models to put together an ordered set of criteria for good predictive models in the reliability area.

Accordingly, the problems addressed in this section are ordered in terms of their priority of need.

● PROBLEM: Need for data.

KEY FACTORS:

- A need for execution time data rather than calendar time data.
- better planning of data collection efforts (this should be done in conjuction with reliability researchers)
- need cost impact data
- need data on resources used in identifying and collecting the data.

RECOMMENDED APPROACH:

Detailed studies should be undertaken to:

- specify what data should be collected and how.
- study should be reviewed by the principal researchers in the field.

TIMING:

Near future (2-3 years). Prospects for success are good.

● PROBLEM: Need a comparative study of existing reliability models.

KEY FACTORS:

- an analytical/anatomical comparison
- predictively comparison
- a physical interpretation of the parameters of the models
- simplicity and ease of understanding and communicating in each of the models.
- range of applicability.

RECOMMENDED APPROACH:

A serious analytical and empirical comparative study to ensure a correct interpretation of the models and the assumptions that have used been used in creating the models.

TIMING:

Such a comparative study should start in the near future and possibly could be completed by 1980.

● PROBLEM: The need to validate the assumptions used in existing models.

KEY FACTORS:

- the independence assumption of failure time
- assumption of an exponential distribution being the underlying relevant statistical distribution.

RECOMMENDED APPROACH:

A study should be carried out.

TIMING:

Near future (2-3 years). The success in this endeavor would depend considerably on the availability of data.

● PROBLEM: Relationship between test and operational environments.

KEY FACTORS:

- the effect on the reliability measures.
- how to construct an appropriate test environment.

RECOMMENDED APPROACH:

Research in the fundamental areas.

TIMING/PROSPECTS:

Time of conclusion is not clear. This appears to be a difficult problem.

● PROBLEM: Relationship between program structure and reliability (including combinatoric relationships).

KEY FACTORS:

- modern programming practices
- module switching (N-th order Markov processes)
- information-theoretic approach

RECOMMENDED APPROACH:

Further research

TIMING/PROSPECTS:

Medium term (3-5 years).

● PROBLEM: What quality performance measures are meaningful and useful? What decisions would be supported? How might management decisions affect selected performance measures?

KEY FACTORS:

- availability
- cost impact measure
- predict project completion
- tradeoffs between quality measures and time/cost.

RECOMMENDED APPROACH:

Manager survey study.

TIMING/PROSPECTS:

Near term (1-2 years). Propects for success are good.

● PROBLEM: Getting software reliability concepts accepted and used.

KEY FACTORS:

- selling--convincing managers that these concepts and techniques are useful.
- integration and simplification of concepts.
- adapting reliability as a system requirement. One possible approach is that reliability should be considered as one of the elements in an evaluation of a proposal.

Other recommendations are included under the other problem areas.

TIMING/PROSPECTS:

(2-5 years). Contingent upon success in other problem areas.

● PROBLEM: What sort of error taxonomy is useful?

KEY FACTORS:

- need an end-use orientation classification scheme
- need to collect data
- Is a multi-variate model needed to handle error severity classes?

RECOMMENDED APPROACH:

Conduct a study on planned uses of error data to develop an appropriate classification scheme.

TIMING/PROSPECTS:

Medium term (2-5 years). Prospects reasonably good.

● PROBLEM: How is changing technology going to affect software reliability measurement?

KEY FACTORS:

- microprocessors
- networking

RECOMMENDED APPROACH:

- Augment Rome Air Development Center microprocessor study
- initiate networking study

TIMING/PROSPECTS:

Near term (2-5 years). Start now. Prospects are good.

The Group dealing with LIFE CYCLE SOFTWARE MODELS AND METHODOLOGIES FOCUSED ON THREE MAIN AREAS:

- Initial answers to the questions posed by AIRMICS.
- Some tentative early definitions.
- Some recommendations for further work.

Question No. 1. What needs to be known to understand the development process better?

- activities - the relationship between the the activities, the flow between the activities, and the products coming out of the activities, for example design for maintainability.
- Measurement quantities. We need all the classes of the measurement quantities -- resource consumption measures, rates of accomplishment, or progress, and quality metrics i.e., a capability to measure these actual quantities and relate them to the accomplishment that is being made, measures of progress, quality, productivity, as well as just resource consumption in accomplishment of time-related milestones.
- We need real data for each of these activities and measurable quantities.
- A full analysis is possible now. It requires a concerted effort by a team of experts spanning the disciplines involved in the total life cycle.

Question No. 2. What information should be collected about the process, product and their interactions?

- The answer to this question is in part an answer to question 1, above. But we also need to know for what purpose ? The answer to this would appear to be to model and use for:
- Management, control and evaluation.
- Improvement of the process.

Neither can be done adequately in order to achieve full life cycle effectiveness without an adequate understanding.

Question No. 3. What experiments and evaluations need to be performed?

- controlled experiments should not be used because:
  - We cannot isolate the problem.
  - Extrapolation is not possible for small projects.
  - It is too expensive.

The recommended approach should be to conduct studies, gather data and tie back to analysis based on common definitions and standard measurements, techniques and models.

Question No. 4. How can we capture program complexity?

- There are a number of existing investigations now ongoing.
- We should monitor these carefully.

Question No. 5 . How can we convince managers to experiment on thier software projects?

- A straight answer is -- Don't attempt to convince them to experiment. The real question is how to persuade managers to collect data for others to use. (There is a real problem here because of fear that collecting the data will be used against managers to show that they wasted resources, that they didn't manage effectively.)
- A partial answer to this question is:

  (1) feedback. It should be a two-way flow. Data is captured from the managers to measure progress and to help improve the process, then they should get the benefit of the feedback to help them manage better.
- A second partial answer is:

  (2) automate the collection effort; make it painless to do so that it doesn't interfere or take away from the effort that is devoted to the project.

Question No. 6. What progress is being made on the transfer of learning from project to project and within organizations?

- Not much. But workshops, such as the Software Life Cycle Management Workshop and conferences on the subject of software engineering help. At least they bring to the forefront an awareness of a lack of transfer of learning from project to project, within and between organizations.

- In order to leave an effective transfer of learning, definitions and common terminology are essential.

Question No. 7. What are the criteria for a good predictive model?

- Parameters of the model should be:
  (1) based on a standard set of definitions. For example, time, effort, manpower, end product (quality of source)
  (2) parameters should be measurable.
  (3) parameters should reflect the environmental needs, not product attributes.
- Each model should adequately cover factors causing variation in model output.
- The set should be adequate to cover the entire life cycle.
- A clear understanding of the domain of applicability.
- Should support management activity i.e., management activities of planning, control and evaluation should be able to be displayed against each function together with the associated function time and cost of the activity. The table below shows this concept.

	Plan	Control	Evaluate
Function	. .	. .	. .
Cost	. .	. .	. .
Time	. .	. .	. .

Question No. 8. How can statistical and analystical models be combined?

- We should not concern ourselves with this. We consider it non-issue and it would more appropriately be left to individual researchers to apply the appropriate academic tool in a solution to the problem at hand.

Question No. 9. Is there a need for a standard set of models?

- Yes.
  (1) to cover the life cycle.
  (2) for different environments.
  (3) to handle factors involved in the process, e.g., resources, reliability, growth. There would be all kinds of time phases in order to take care of overlapped activities, for example.

Question No. 10. Do we need a new set of models, or are there already models that adequately satisfy the need?

- The answer to this seems to be that there already exist adequate models flow which to build upon, but we need to have a cataloging of these models to define their capabilities, the underlying assumptions and the validity of the results that they will yield. We need an extension in the terms of the agreed upon definitions and we need an extension of these models to satisfy the criteria in response to question No. 7, that is, it should satisfy the management activities matrix -- the planning, control, and evaluation, and should identify the functions, the time and the cost of each of those activities.

Question No. 11. What are the components of an overall software engineering methodology?

- there should be a statement of objectives
- how to do it
- a means for quantitative evaluation (What is being done and achieved)
- manageable
- executable by normal people in a normal environment
- a definition of the range of applicability

Question No. 12. Where should software technology be going?

- In terms of products, which would include microprocessors and their support software, the technological thrust should be to provide complete functional specifications and to be able to tally a defined set of standard interfaces.
- A continuous Life Cycle in which each activity fully supports all those that follow.
- There should be high level environmental objectives and parameters.

Question No. 13. "Are there standardizable methodologies?" supposes that there is a need within the Army and within the Department of Defense for standardization. This implies a taxonomy. There would be three dimensions involved in the answer. The three dimensions are:

- Environment
- Activity (phases)
- What is being addressed (for example, cost, reliability, performance).

Question No. 14. The effect of software engineering requirements and environmental factors are fundamental to this. The answer to this is adequately covered in the responses to the earlier questions.

Question No. 15. How can we characterize the methodology? The answer to this is again the answer to 13.

Question No. 16. Is there a way to measure the effective life cycle.

- The answer appears to be yes and, as an example, the evolution dynamics of Belady and Lehman is an approach that provides considerable insight.

Question No. 17. When should a system be redone? We need both static and dynamic indicators.

- Study is needed in this area to classify and catalog what these indicators should be.
- Current practices in industry may be helpful in this classification action.

With respect ot the general questions, what are the priorities? We should go after solving those issues relating to life cycle management of software. It appears that the number one priority is to establish a common set of definitions. All of the others are important but they are hindered by a lack of required definitions. All these problems that have been identified solvable in the next five years. Most of the ones that have been identified and commented upon are solvable in the next five years if the effort, direction and resources are focussed on their solution.

What should next year's questions be focussed on?

- Agreement on a common set of life cycle components, or phases.
- Status of the current research now on/going.

TENTATIVE DEFINITIONS

1. Large: A software project is large if it involves at last two separately managed groups.
2. Life cycle: The life cycle of a software project encompasses all the activities from first formal conception until final abandonment. When we refer to "life cycle" of an activity/phase, it must always be qualified to some extent, (for example, we should refer to the development cycle protion of the software life cycle).

RECOMMENDATIONS:

- Data collection and definitions. Recommend setting up a standing committee for software life cycle management data collection, involving identification and definition. This might be done by some organization such as AIRMICS.
- Establish a catalog of methodologies and models. There should be a group to identify, to improve and to recommend adoption of appropriate methodologies and models.

LIST OF NEEDED DEFINITIONS:

1. Productivity
2. Life cycle phases
3. Lines of code
4. Complexity
5. Software maintenance (modification, enhancement, debugging, error fixing)
6. Error
7. Reliability (quality metrics)
8. Man month (effort measurement)
9. Software (system)
10. Verification, validation.

CONCLUSION:

Good progress has been made in this software lifecycle management workshop. It is felt that the life cycle management workship is an important forum focussing ideas for improvement in future action. The results observed in this workship appear to be fruitful and encouraging with respect to what was identified and pointed out in the first software life cycle workshop a year ago. Continuation appears to be clearly indicated.

APPENDIX TO REPORT GROUP III

(Submitted by Leon G. Stucki)

I. PROGRAMMING ENVIRONMENT

Background

● Paradoxically, the software community has automated everyone's work except their own.

● Isolated tools and techniques have been developed.

● Software is still extremely labor intensive.

● Productivity improvements in software construction has not kept pace with hardware.

● Software is rapidly becoming the limiting factor in large systems.

● An automated programming environment with an integrated set of tools for the management, control, testing, and documentation of each stage the evolving software offers a means for greatly reducing software costs and improving software quality.

● Tools within an automated programming environment should include:

- Source language level interpreters, for statement-at-a-time execution and tracing
- Compilers, for both program debugging (e.g., extended syntax checking and user feedback) and optimizing
- Cross-compilers
- Text editors, CRT terminals
- Configuration management aids
- Automated verification and testing aids
- Interactive debugging aids

● Functions provided by a programming environment should include:

- Mechanisms for controlling and documenting the communication process between users-analysts-programmers-managers.
- A central repository, with supporting data bases, for configuration management and control of all documentation and evolving program representations (design and code).
- Quality assurance mechanisms for checking adherence to project standards.
- Automated error collection and reporting in support of both quality assurance and configuration management.
- A respository of test data and test results traceable to user acceptance/test criteria.

Current State

● Exaggerated claims have been and continue to be made for isolated tools and techniques.

● Most program development is done with severely inadequate tools.

● A compiler is frequently equated with a programming environment. (In reality, a compiler constitutes only one small, albeit important, component of an automated programming environment).

● Much manual drudgery still prevails in most current programming environments.

● Errors once discovered and removed may reappear due to the manual processes currently employed in building today's systems.

● Management visibility into the progress of software development is woefully inadequate.

● Experimental use is being made of selective "proven" tool and technique concepts not yet widely available (e.g., static and dynamic analysis aids).

Future Trends

● Programming environments will be designed

Question No. 14. The effect of software engineering requirements and environmental factors are fundamental to this. The answer to this is adequately covered in the responses to the earlier questions.

Question No. 15. How can we characterize the methodology? The answer to this is again the answer to 13.

Question No. 16. Is there a way to measure the effective life cycle.

- The answer appears to be yes and, as an example, the evolution dynamics of Belady and Lehman is an approach that provides considerable insight.

Question No. 17. When should a system be redone? We need both static and dynamic indicators.

- Study is needed in this area to classify and catalog what these indicators should be.
- Current practices in industry may be helpful in this classification action.

With respect ot the general questions, what are the priorities? We should go after solving those issues relating to life cycle management of software. It appears that the number one priority is to establish a common set of definitions. All of the others are important but they are hindered by a lack of required definitions. All these problems that have been identified solvable in the next five years. Most of the ones that have been identified and commented upon are solvable in the next five years if the effort, direction and resources are focussed on their solution.

What should next year's questions be focussed on?

- Agreement on a common set of life cycle components, or phases.
- Status of the current research now on/going.

TENTATIVE DEFINITIONS

1. Large: A software project is large if it involves at last two separately managed groups.
2. Life cycle: The life cycle of a software project encompasses all the activities from first formal conception until final abandonment. When we refer to "life cycle" of an activity/phase, it must always be qualified to some extent, (for example, we should refer to the development cycle protion of the software life cycle).

RECOMMENDATIONS:

- Data collection and definitions. Recommend setting up a standing committee for software life cycle management data collection, involving identification and definition. This might be done by some organization such as AIRMICS.
- Establish a catalog of methodologies and models. There should be a group to identify, to improve and to recommend adoption of appropriate methodologies and models.

LIST OF NEEDED DEFINITIONS:

1. Productivity
2. Life cycle phases
3. Lines of code
4. Complexity
5. Software maintenance (modification, enhancement, debugging, error fixing)
6. Error
7. Reliability (quality metrics)
8. Man month (effort measurement)
9. Software (system)
10. Verification, validation.

CONCLUSION:

Good progress has been made in this software lifecycle management workshop. It is felt that the life cycle management workship is an important forum focussing ideas for improvement in future action. The results observed in this workship appear to be fruitful and encouraging with respect to what was identified and pointed out in the first software life cycle workshop a year ago. Continuation appears to be clearly indicated.

and selectively implemented.

● Hardware manufacturers will provide machine/language dependent environments.

● Techniques will developed to isolate language and operating system dependencies as much as possible (e.g., attempts will be standardize the interfaces).

● HOL standardization within DOD will make it possible, for the first time, to achieve a rich program development environment accessible to larger numbers of people.

● The "National Software Works" concept will provide valuable knowledge on the success and failure of many of these concepts in a distributed environment.

● Additional textual and graphical techniques and automated tools will be developed for representing, documenting, and controlling the iterative nature of early phases of program development (i.e., requirements and design).

● An integrated framework will be developed for applying numerous analytical techniques (e.g., consistency and completeness of testing, formal proof techniques for selected system components, static/dynamic/symbolic analysis of subsystems).

● The theory of testing will receive more academic attention than in the past.

● Improved techniques will facilitate the certification and recertification process of future systems.

II. LANGUAGES AND ARCHITECTURE

Background

● Testing has historically been and continues to be very costly.

● The concept of built-in test circuitry in hardware is widely acceptable and increasing in application. Similar approaches can and should be applied to software.

● Top down elaboration and refinement of acceptance/test criteria can be generated in parallel with system development.

● Acceptance/test criteria when incrementally developed and included in source code via special comments (or test assertions) have been shown in experiment to improve software quality and reduce testing time.

o This concept can be used in conjunction with both dynamic run-time analysis systems and formal verification systems.

Current State

● Several prototype systems have been built or designed; however, none are currently operationally used.

Examples:

- Stucki's experiments with a PL/1 prototype system at UCLA.

- University of Texas Gypsy programming system.

● Current language work on DOD-1 has provided "an opening" through a very fuzzy assertion concept.

- An assertion statement has been provided in the language, but its syntax and semantics and use are unspecified at this time.

Future Trends

● Further procedures will be developed for:

- specifying acceptance/test criteria during the system requirements phase, and

- refining the acceptance/test criteria throughout the subsequent design and construction phases.

● Language work will provide new and more powerful constructs for expressing self-test and monitoring concepts.

● Automated tools employing these concepts will be able to greatly improve the testing and maintenance processes.

III. FRAMEWORK FOR MODELLING AND SIMULATION

Background

● Modelling and simulation have been used widely by various analytical disciplines.

● The models of the various disciplines

have generally been incompatible.

- There is a need to provide a framework and data base mechanism for controlling and accumulating knowledge of a given system gained through various modelling and simulation activities.
- Executive and utility functions include:
  - Model/Data input preparation and storage
  - Assistance in the creation and maintenance of interfaces between models
  - Assistance with output report preparation

<u>Current State</u>

- The Air Force is currently studying the requirements for at least one such system (i.e., General Modelling System project under ICAM).
- Other industry efforts in CAD/CAM (computer aided design/computer aided manufacturing) are exploring this area.
- Interfaces to hardware are increasing as digital computers replace analog devices.

<u>Future Trends</u>

- Prototype systems will be built and studied. (Application specific systems will be available.)
- Systems will support hierarchies of models as well as interdisciplinary interfaces.
- Increased interaction will also be involved with actual sensor systems.

IV. LIFE CYCLE MANAGEMENT MEASUREMENT METRICS - MEASURES & EMPIRICAL STUDIES

Chairman: L.A. Belady
Thomas J. Watson Research Center

PANELISTS

Bill Curtis	Sandra Mamrak
James L. Elshoff	Thomas J. McCabe
Maurice Halstead	James A. McCall
Maryann Herndon	Isao Miyamoto

Alan N. Sukert

MEASURES AND EMPIRICAL STUDIES

Summarized by
L. A. Belady

IBM Research

Following are the panel's observations and recommendations on the use of metrics to improve the understanding and management of the software development and maintenance process.

We see it encouraging that, compared to last year's workshop, the present papers are more evaluative than speculative. This trend should continue, moreover, emphasis should be on *collecting facts*. During its short history, software sciences have been characterized by a large number of ideas, techniques and tools proposed, with the result that there are now more ideas available than necessary or possible to apply. Yet there is a definite shortage of *tested ideas*. Institutions and universities have been developing new techniques and approaches to improve the process. At the same time builders of large systems, and those who are in charge of maintaining these extremely complex man-made objects, are still forced to use antiquated methods. Neither party is at fault: the problem is that there is no way to demonstrate whether an idea is viable and whether it will beneficially influence the development process. Thus gathering facts about the software, and about the process developing it, is the most important next step, without which there is no hope for successful transfer of technology.

But merely gathering facts is not enough. Facts should be structured and appear in a format that permits the comparison of systems, situations, processes. We are convinced that there already exist proposed and quite *promising metrics* which, although applied so far only to limited samples, showed interesting results. Consequently, instead of inventing additional metrics and thus increase unnecessarily the variety of available approaches, we must broaden the basis on which existing metrics are applied. Coming to mind are proposals by Halstead, McCabe, the Belady-Lehman measures performed on large systems, and others found in the literature. We should concentrate on a handful of the most promising approaches, align them with each other and standardize.

We may gain confidence in these metrics by examining their usefulness in three roles. One role is to extract generally valid laws about the behavior of large systems, large projects and that of programmers. The second is to predict the evolution of the very project or system being measured. The third factor is psychological, namely the feeding back of the observations, and predictors based on them, to the people who created or caused the process to happen. For example Elshoff of GM Research found it often useful to make visible the otherwise invisible object, the program itself.

In fact, we are talking about the existing and sufficiently validated metrics, many of them already applied, successfully and independently of hardware characteristics, to monitor the development of new projects and the maintenance of older systems. Clearly, if we want to thoroughly and carefully evaluate the usefullness of an approach, we cannot rely exclusively on computer scientists and software engineers: other experts must also participate. Psychologists for example are trained to evaluate complex situations using the rigorous and well established methods of experimental design and statistics. What we, therefore, propose is a multi-disicplinary approach toward the evaluation of the *already existing* but sparsely used ideas in order to weed out bad approaches, and to gradually improve and refine the ones, whose potential use for any or all of the above mentioned three roles is immediate. It is also important that we restrict measurements to a handful of observables, such as Halstead's operators and operands, and then deduce the other attributes such as portability, modifiability, maintainability from primitive metrics.

This leads us to the most difficult attribute of the process *productivity*. While we believe in its importance, we cannot accept its current unit of measure, namely lines of code per unit time or man-month. The reason for this disbelief is twofold. First, most programmers will be just modifying and maintaining already existing programs. Secondly, in the future more and more new programs will be constructed out of off-the-shelf components. Whether in modification and maintenance, or system composition from larger components, the line of code measure of productivity immediately fails.

We measure productivity for perhaps two reasons. One is to monitor costs and the second is to predict the resources necessary for the development of new products. But the lines of code generated is just one of the many components of the total cost. Clearly, the quality of development influences the cost of maintenance and modification, to be performed over a long period of time. Thus, if we want to measure productivity at all, then it has to be *combined with* a metric capturing *quality*. Only then can we have a solid measure for prediction, as well as for comparison of different systems and projects.

As already indicated before, the major obstacle to progress and improvement is the difficulty to transfer technology, i.e., ideas into real-life production situations. Take, for example, a methodology, which in a small environment, and mostly by the inventors of the method, is believed to significantly better than the currently used methods. First, the new idea must fall on fertile ground. This means that not only do the receivers of the idea have first to be *educated on the novelty* being proposed but their mind has to be open and well informed about the large variety of other alternatives. Only then can a dialogue develop and factual evaluation take place before commitments to the new method. Second, the dominant factor in successfully transferring technology is that a new proposal must demonstrate its viability by facts. Otherwise there is no change to transfer methodology.

Project management is right in refusing untested methods, untested ideas and techniques. Proposals must be demonstrated to be beneficial to the project. The major problem now is that we do not have any place, any *forum* or any organizational entity *where* at a reasonable scale, *methods* can be tested and their viability *demonstrated*. Thus the creators of ideas remain frustrated. They never see the utility of the ideas on which they work so hard. Yet they should know which ideas make sense and which not. It is better to know that an idea does not make sense in real life than to remain with the uncertainty about its value and then blame the developers for not implying something which is supported only by speculation.

This leads us to the problem of where to generate facts about new proposed methods, techniques, tools and other novelty. Why does the Army not set aside resources for the sole purpose of validating the ideas created inside and outside its own organization? Unusable novelty would rapidly disappear while usable ones find their way into development, thus improving the overall quality of the software life cycle. But even without such an ideal, separate organization, systematic gathering of facts on real projects would still tremendously improve the learning process and encourage the flow of information about techniques and tools because measuring with agreed upon and aligned metrics facilitates comparison of methods applied, projects managed and software produced. Again, emphasis must be on facts, and on measured and comparable facts.

At the workshop it became also clear that our universe of discourse is not homogeneous any more. There is no such thing as a typical program, typical project, or typical situation. What is, indeed, badly needed is the taxonomy, the *classification* of may aspects of the life cycle. Before we even start measuring, we must know precisely what we measure and what we compare against. We must set up the metrics and the measurements according to whether they are about a small, medium or large size project, whether we measure an on-line, interactive or batch system or its development, and so forth.

The other important aspect of taxonomy is that we have to recognize the *limits of validity* of all the metrics and models which we apply. Similar to physics, life cycle management must also have scaling or model rule effects. Every metric, every measurement method has its domain of validity. Beyond this domain of validity, one may have to live with false results or else must adjust the metric with some suitable parameters. Moreover, classification in the software development and maintenance must be along many dimensions: development, specification and standardization of error classes, categories of programs, processes and so forth. Briefly: taxonomy is one of the most important prerequisites to good measurement and then to good and valid comparison of the measured objects.

A specifically important case for precise *definitions* and standardization is the case of milestones. In general, it is desirable to subdivide a project in two dimensions. Time is one of the dimensions: one would like to see the precise transition point from one phase of the life cycle to another; for example the point defined by the end of design and the beginning of implementation. The other dimension is the product itself, namely its decomposition into major components. It is also necessary to decompose the cost estimates and then the actual cost items along both dimensions. Thus we must find ways to precisely specify and mutually agree upon this two dimensional grid which is applied over the total project in the time and the product domains. Without agreed upon definitions, such as the events which specify the transition from one phase of the life cycle to another, milestones have absolutely no meaning and their use probably causes more confusion than allow for sound monitoring and comparison.

A few words about tools. We mean here techniques and instruments which are necessary to extract the facts and then form metrics: the *tools of data collection*. Interestingly enough we all agree that there already exist built-in methods, and techniques, which continuously collect data which in turn are never interpreted or used. In fact we believe that one could start immediately at practically zero cost to gather data without the additional building of new tools. Compilers are an example. During compilation large amounts of significant data are collected, but after having produced the code, the contents of internal tables become discarded. Intelligent use of already generated data as a basis for metrics and meaningful comparisons within and between the different systems and projects would be an almost zero cost activity. We invite the Army to first look around for already existing tools and data already being gathered before a vast and expensive tooling up of projects starts.

We consider the order of importance of things to be done as implied in the order above. Almost all proposals are doable within the next five years. An exception is perhaps the error taxonomy which should be a research effort. We see also quite dark with respect to a good comprehensive and sensible productivity measure. The Workshop Chairman wanted some questions for next year's workshop. Well, Barry Boehm proposed, the best such question: "What happened to the recommendations of the previous year?" We hope for the best.

We do not propose here specific research activities now. Rather, we call attention again to the *multi-disciplinary* approach. The role or psychological research, particularly its role in evaluating proposed methods and techniques, should be significantly increased. Experts from sociology and management sciences should also play an increasing role in life cycle related research. Also, the use of time series analysis must be introduced. But in any case, future research should be based on actual data, moreover, on aligned data. As long as scattered research groups all define and interpret their own data, or use other people's unaligned data, we cannot expect that transfer of knowledge from one place to another be possible. In fact, we propose that a *central data base and clearing house* for data be established within DoD in order to provide badly needed factual information as a basis for coordinated research. We understand that the Rome Air Development Center will soon be ready to play this extremely important role. Also along centralization, we propose the rigorous definition of the following ten most important terms: the six or seven phases of the life cycle: requirement, specification, design, etc: complexity, quality, productivity (and probably all the "abilities" which are so heavily used, yet never defined). Some efforts already exist within the Air Force, the Army and GTE Corp. Without such definitions technical people in large organizations are forced to use local definitions or take as source the trade magazines and professional literature.

In summary, we believe that a *drastic and massive shift to fact finding and to organizing the knowlede we already have*, must characterize the next years to come.

V. POSITION PAPERS

LIFE CYCLE MANAGEMENT METHODOLOGY
DYNAMICS - THEORY

"Modeling, Measuring & Managing Software Cost"
John R. Brown, Boeing Computer Services Company

Improving the Signal/Noise Ratio of the System Development Process"
Melvin E. Dickover, SofTech, Inc.

"A Step Towards the Obsolescence of Programming"
Harvey S. Koch, University of Rochester

"A Contingency Theory to Select An Information Requirements Determination Methodology"
J. David Naumann & Gordon B. Davis
University of Minnesota

"A Life-Cycle Model Based on System Structure"
Francis N. Parr, Imperial College of Science and Technology/England

"The Implications of Life-Cycle Phase Interrelationships for Software Cost Estimating"
Robert Thibodeau and E. N. Dodson
General Research Corporation

MODELLING, MEASURING AND MANAGING SOFTWARE COST

JOHN R. BROWN
Boeing Computer Services Company
Seattle, Washington 98124

Abstract

An appraisal of past experiences relevant to achieving awareness of the cost of software is provided in terms of personal recollections about the "good old days". The difference between what we plan to do and what we really do in developing software is discussed and identified as a significant source of the cost problem. A striking similarity between the properties of computer programs and the characteristics of the software development process is suggested. Application of computer program analysis tools to support detailed evaluation of the development process is proposed. Some potential benefits regarding improved understanding of software production costs are discussed and related to possible modification of current software procurement practices.

Introduction

A few years ago a good friend of mine wrote a very interesting letter entitled, "Random Thoughts on Software Integrity, or, Nostalgia for the Good Old Days". Among other things, the letter served to refresh my recollection of the good old days while stimulating something akin to at least a rough comparison of then and now.

In order to get my thoughts straight I found I had to determine approximately when "then" stopped and "now" started. Having worked continuously in one way or another with the production of computer programs since sometime in 1960, it is perhaps meaningful to identify some point in time which separates the good old days from whatever one might call more recent times. For me such a delineation comes at approximately the midway point, that is in late 1967 or early 1968. As I pursued this train of thought, I became aware (or finally admitted to an awareness) of certain facts. I believe that my findings are relevant to any discussion on the claimed cost and difficulty of developing software, and I hope to demonstrate relevance in the following paragraphs.

In Retrospect

At some point early in my rambling thought process on the subject, I found myself hard put to answer the question, "When did things start to go wrong?". After considerable soul searching, I was able to settle upon a fairly specific time period during which I had begun to understand that there were certain problems associated with software development. It is especially important to note that I have not intimated that software development was problem free prior to 1968 and plagued with problems thereafter. I have simply concluded that in late 1967 and early 1968 something happened which prompted (in me) a rather keen (but previously non-existent) awareness of some software development problems. I found it instructive for my purposes to attempt to identify whatever it was that was special or different about that point in time and have related memorable characteristics below in no particular order.

- I had recently been given my first real fiscal responsibility within a major (large scale) software development activity.
- I was, prior to that time, only remotely aware of the fact that software is developed for a customer. At this point I began to be exposed to the needs, hopes, fears, and frustrations of a customer on a regular (almost daily) basis.
- I was asked for the first time to deliver a large program to a customer. In particular, it was a program which contained a number of large, integral elements about which I personally knew little or nothing.
- I become aware of the existence of the incomplete requirement specification. More importantly, I became convinced that it played a critical part in a supposedly "formal" software development process.
- I was asked (albeit very indirectly) and not in so many words to compromise "intangible" quality in favor of tangible, timely (on schedule) delivery. I was subsequently required to "explain away" problems or relate them to known errors and lack of specificity in the requirement specification.

I am not at all sure which of the above was most instrumental in changing my view of what software development is all about. Perhaps more important than any of the individual items was ultimately the frame of mind which came from living through and coping with these new (for me) experiences. The most striking characteristics of my new frame of mind was a strong realization that software development costs a lot of money. Coupled with this was the growing feeling that there were some customers who were hard to convince that they had gotten their money's worth, especially if the software did not work exactly as expected or better. Unfortunately, specifying exactly what is expected of software has proven to be at least as difficult, if not moreso, than specifying things in general [1,2,3]. In fact, a great deal of the thinking that has been given to the cost and quality of software has concerned the difficulties inherent in specifying intent (i.e., requirements), testing and demonstrating satisfaction of those requirements, and providing for full and timely communication between software developers and users [4,5,6,7,8].

About Software Life Cycle Cost

So far, I have simply related some of the details and subsequent conclusions derivable from a conversation with myself about the good old days. It is perhaps apparent that I have more or less dispelled the notion that there really were any good old days, but rather that there was a time when, for a number of reasons, programming was fun and my worries were few.

As is often the case with my rambling thoughts about software development, however, I eventually found myself taking a hard look at the development process from a different but potentially useful point of view.

Most, perhaps all, people who claim to know how software gets developed roughly view the development process as a serial one which includes the following (or equivalent) events:

1) Concept (Requirements) Definition
2) Detailed Requirements Specification
3) Preliminary Design
4) Detailed Design
5) Code and Debug
6) Checkout
7) Test Planning
8) Test Execution
9) Test Evaluation
10) Acceptance and Use
11) Maintenance (modification) and Re-Test (as required)

There are many variations which are more or less equivalent to the above and apparently many different impressions about the proper ordering of the events [9,10,11,12,13]. For purposes of this discussion we can assume that some major software development activities proceed through phases leading to the above sequence of events with the usual iterative occurrence of events 4 through 9.

It appears that we can be comfortable with (and readily accept) a not-quite-accurate picture of the development process to which we attribute serial orderliness and implicit continuity. Further, and more important, we are hard pressed to learn very much about the real costs of software development until we can shed the notion of a fixed sequence of events and come to grips with the complex and highly dynamic interaction of these events which is characteristic of much software development activity. For instance, if we were to draw a flow diagram of the development process (both "before" and "after"), we could well see the kind of contrast illustrated in Figures 1 and 2.

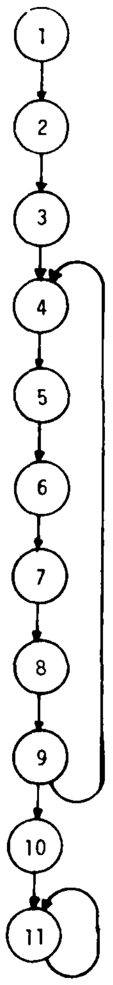


Figure 1. A "Before" View of Software Production

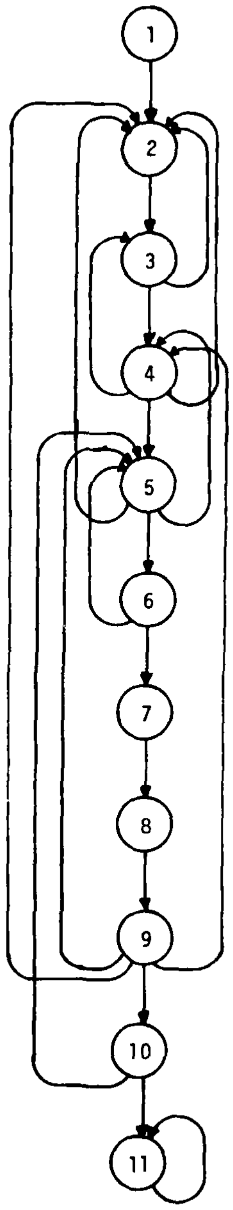


Figure 2. An "After" View of Software Production

Towards Understanding Production Cost

It is probably not through sheer coincidence alone that we use the word "program" for very large software development projects. If put to the test, we can find many similarities between the properties of the end item (i.e., the computer program) and the "program" or process through which it was produced. We have labored long and hard to develop techniques and tools [14,15,16,17, 18,19,20] which provide valuable insight into the intricate and highly complex interactions of the components of computer programs. We have rigorously applied such tools in the study of computer program efficiency and have saved countless thousands of seconds of computer time. Unfortunately, we have not done nearly so well in shaving seconds off the time required for development of complex software systems. Perhaps it is time that we seriously considered application of such tools in investigation of the other kind of programs. Maybe then we could begin to develop a better understanding of how software really gets put together and where, in the process, the high cost of software truly lies and where efforts to reduce excessive cost can and should be concentrated [21,22]

A Proposal

I am particularly intrigued with the potential application of program path analysis and usage monitoring tools [23,24,25] to help achieve a more thorough understanding of what really takes place deep within the development process. I suggest, for example, that something along the following lines could be done. First, we devise a project (program) plan which possesses all the conditional branching (go back and redo, etc.) characteristics which are illustrated in the previously defined schematic labelled AFTER. Then, as we think of the programmatic events much like the segments or units of code within a program, we may define a convenient notation and procedure for developing path-like sequences which alone or in combination represent potential modes of traversing the network [25,26,27]. Now if we also took steps to assign time-to-complete (TTC) and cost-to-complete (CTC) values to each of the eleven previously identified events, it is possible to estimate significant project cost and schedule variations as a function of our "best guesses" at the way project work will actually proceed. We must be careful however to consider the following:

1) TTC and CTC values are probably variable and are functions of many factors including 1) state-of-the-art of the specific technological area, 2) level of experience and expertise of available personnel, 3) event count (i.e., it may or may not be cheaper to write detailed requirements for the third time than for the second time), and 4) event predecessor relationships (i.e., it may cost more (or less) to revise or redevelop a preliminary design after extensive testing has been completed than before).

2) Where appropriate perhaps in accordance with existing procurement regulations or customer-contractor agreements) those branching characteristics which are to be disallowed must be taken into consideration. For example, there is at least a common sense requirement that test evaluation is not followed by preliminary design more than n times.

3) It may be important to avoid the premature conclusion that the path 1-2-3-4-5-6-7-8-9-10-11 obviously presents the most appealing cost picture, because it may well be the most unrealistic path in terms of potential development activity.

Finally, we might look briefly at one possible derivative of the approach briefly outlined above. For example, consider the possibility of a customer somewhere who:

1) has a problem to solve which requires procured services for the production of a software system, and

2) is willing and able to establish constraints of the type mentioned above in 2.

We can then conceive of competitive contractors who submit proposals consisting of the usual technical volume (i.e., background information, statement of work, technical approach, related experience, facilities, etc.). Consider, however, a very different kind of management and cost volume which presents a predetermined number of cost and schedule proposals corresponding to at least the high likelihood paths through the network. The availability of tools to support this kind of effort is assumed, since I personally know of a large number of computer program analysis tools which would require only minor modification to provide network analysis, path generation and display capabilities. These tools are described at length in the literature primarily addressing subjects of automating software testing, monitoring and measuring the thoroughness of testing, and static analysis of program structure [6,7,10, 14,18,19,23,24,25,26,27]. One might expect then that we could readily acquire a capability to monitor project-program operation and subsequently compare expected-versus-actual operational experience with life cycle costs.

Perhaps then if we:

1) develop and experiment with an approach to procurement, program planning, cost estimation and performance assessment something like that described here,

2) use tools to maximum advantage to support more objective yet precise consideration of pertinent cost factors, and

3) try to be as honest as possible with each other about what all this means,

then I expect we will begin to gain the kind of insight needed in order to get a handle on and do something about the real problems relevant to the high cost of software development [28,29,30]. It is possible that, with practice, using the cost modeling technique, we could get pretty good at estimating at least upper and lower bounds on project cost. It's possible also that in our quest for truly improved software engineering practice [29] the modeling technique can help us give more careful attention to the real cost payoffs from modern techniques and tools and further promote their judicious and cost effective application to future development activities. We might even be able to stop talking about the good old days as if they were some time in the past.

References

1. Gunning, Robert, How to Take the Fog Out of Writing, Dartuell Press, Inc., Chicago, Ill., 1962, 64 pages.

2. Hartman, P. H., and Owens, D. H., "How to Write Software Specifications," Proceedings of Fall Joint Computer Conference, 1967, pp. 779-790.

3. Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973, pp 48-59.

4. Boehm, B. W., Brown, J. R., et al., Characteristics of Software Quality, TRW Series of Software Technology #1, North Holland, January, 1978, (Previously published as TRW Technical Report No. 25201-6001-TU-00, December, 1973).

5. Brown, J. R., DeSalvio, A. J., Heine, D. E., Purdy, J. G., "Automated Software Quality Assurance: A Case Study of Three Systems," Program Test Methods (Ed. W. C. Hetzel), Prentice-Hall, 1973, pp. 181-203.

6. Brown, J. R., and Hoffman, R. H., "Evaluating the Effectiveness of Software Verification - Practical Experience with an Automated Tool," Proceedings of Fall Joint Computer Conference, 1972.

7. Brown, J. R., "Improving Quality and Reducing Cost of Aeronautical Systems Software through Use of Tools," Proceedings of Air Force Aeronautical Systems Software Workshop, April, 1974.

8. Kosy, D. W., "Air Force Command and Control Information Processing in the 1980's: Trends in Software Technology," Rand Report No. R-1012-PR, June, 1974.

9. Williams, R. D., "Managing the Development of Reliable Software," Proceedings of the International Conference on Reliable Software, April, 1975, pp. 3-8.

10. Brown, J. R., "Getting Better Software Cheaper and Quicker," Practical Strategies for Developing Large Software Systems, Addison-Wesley, 1975, pp. 131-154.

11. Mangold, E. R., "Software Visibility and Management: Technology," Proceedings of the TRW Symposium on Reliable, Cost-Effective, Secure Software, TRW-SS-74-14, March, 1974.

12. Boehm, B. W., "Software Engineering," IEEE Transactions on Computers, Vol. C-25, No. 12, December, 1976, pp 1226-1241.

13. Schluter, R. G., "Experience in Managing the Development of Large Real-Time BMD Software Systems," Proceedings of AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, November, 1977, pp. 168-173.

14. Brown, J. R., "Software Test Tools: Technology," Proceedings of the TRW Symposium on Reliable, Cost-Effective, Secure Software, TRW-SS-74-14, March, 1974.

15. Brown, J. R., "Why Tools?," Proceedings of Computer Science and Statistics: Eighth Annual Symposium on the Interface, February, 1975, pp. 310-312.

16. Mullin, F. J., "Considerations for a Successful Software Test Program," Proceedings of AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, November, 1977, pp. 68-74.

17. Kessler, M. M. and Kister, W. E., "Software Tool Impact," Structured Programming Series, RADC-TR-74-300, Vol. XIV, May, 1975.

18. Osterweil, L. J., "A Methodology for Testing Computer Programs," Proceedings of AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, November, 1977, pp. 52-62.

19. Reifer, D. J., "Automated Aide for Reliable Software," Proceedings of the International Conference on Reliable Software, April, 1975, pp. 131-142.

20. Stucki, L. G., and Foshee, G. L., "New Assertion Concepts for Self-Metric Software," Proceedings of the International Conference on Reliable Software, April, 1975, pp. 131-142.

21. Black, R. K. E., "Effects of Modern Programming Practices on Software Development Costs," Digest of Papers from the Fall Computer Conference, COMPCON 77, September, 1977, pp. 250-253.

22. Brown, J. R., "Modern Programming Practices in Large Scale Software Development," Digest of Papers from the Fall Computer Conference, COMPCON 77, September, 1977, pp. 254-258.

23. Brown, J. R., and Hoffman, R. H., "Automating Software Development: A Survey of Techniques and Automated Tools," TRW-SS-72-03, May, 1972.

24. Brown, J. R., "Practical Applications of Automated Software Tools," WESCON 1972, Session 21 and TRW-SS-72-05, September, 1972.

25. Krause, K. W., Smith, R. W., and Goodwin, M. A. "Optimal Software Test Planning through Automated Network Analysis," Proceedings of IEEE Symposium on Computer Software Reliability, June, 1973, pp. 18-22.

26. Brown, J. R., and Lipow, M., "Testing for Software Reliability," Proceedings of the International Conference on Reliable Software, April, 1975, pp. 518-527.

27. Brown, J. R., and Fischer, K. F., "A Graph Theoretic Approach to the Verification of Program Structures," Proceedings of the Third International Conference on Software Engineering May, 1978, pp. 136-141.

28. Sukert, A. N., "A Multi-Project Comparison of Software Reliability Models," Proceedings of AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, November, 1977, pp. 413-421.

29. Brown, J. R., "Programming Practices for Increased Software Quality," Software Quality Management, Petrocelli Books, to be published and presented at the Software Quality Management Conference, September, 1978.

30. Brown, J. R., Osterweil, L. J., and Stucki, L. G., "ASSET: A Lifecycle Verification and Visibility System," Proceedings of COMPSAC 78, to be published, November, 1978.

IMPROVING THE SIGNAL/NOISE RATIO OF THE SYSTEM DEVELOPMENT PROCESS

Melvin E. Dickover

SofTech, Inc.

ABSTRACT

Some of the problems of major system developments can be traced to the lack of a rigorously linked chain of documents connecting the operational needs and context to the design given to the implementers. A method of constructing such a rigorous linkage using SADT™ models is outlined in this paper.

PROBLEM

Various problems have plagued the development of large military systems. Among these problems is that the delivered system is not what the user expected and is, in fact, not very useful, even though, in some sense, it works. Another problem is that requirements seem to change rapidly during development, either increasing cost and slipping the schedule or decreasing the usability of the product. The Department of Defense has been trying to address these problems with new regulations (CA-109, 5000.1, 5000.2, 5000.3, etc.); these regulations will improve things. Some important holes in the management process remain, however.

The approach to technical program management taken in this paper assumes some of the troubles in major system acquisitions are due to the following causes:

1. Something gets lost in the translation from stage to stage in the development process. "Noise" accumulates until the product differs appreciably from the user's original version of it. Current military documentation regulations do not prevent this, because they specify form rather than substance.

2. Many of what are called requirements changes are in fact only requirements <u>document</u> changes made necessary by the continual uncovering of old, unchanged requirements not in the document because the requirements definition process was inadequate. [1] [2]

APPROACH

The two causes of the problem are addressed by producing a chain of documents that link the system user's conception of the system to an abstract, implementation-free specification of the system to be given to the implementer. The documents constrain the implementer to produce just what the user needs, but leave him free to trade off the alternative ways of realizing the system.

For this approach to succeed, the following must be met:

1. The documents must be formally, rigorously linked one to the next.

2. For each document in the chain, one must be able to decide what information is or is not supposed to be in it.

3. These documents must relate to the things controlled in the major system acquisition process that the DoD uses.

The documents proposed here are a linked set of SADT models.[3] Before these documents are described, it is necessary to provide a semiformal definition of the terms used in the rest of the paper.

DEFINITION FRAMEWORK

A *system* is a set of interacting "phenomena" of "actual reality." (Quotes indicate primitive terms.)

Questions are predicates about "phenomena."

Answers are propositions about "phenomena."

A *model* is a relation between a set of *questions* and a set of *answers*.

™SADT is a trademark of SofTech, Inc.

A *model* of a *system* is a relation between a set of *questions* about "phenomena" of the *model* and *answers* about "phenomena" of the *system* to some tolerance.

The *purpose* of a *model* is its set of *questions*.

In the definitions that follow, by model it is meant a model produced using SADT. [3]

The *view* of a *model* is the choice of partitioning at the first level of the hierarchy. The *view* is constrained to be consistent with the *purpose*.

The *vantage point* of a *model* is a set of constraints placed on the *view* so that it embodies the abstractions and perceptions of a certain kind of person. (Not only must the *view* satisfy the *purpose*, it must be formed of pieces a person recognizes.)

A *user* is a *vantage point* of a person who interacts with the *system* as it exists in "actual reality."

A *functional model* is a *model* of a *system* from the *view* of a *user* of that *system*.

An *activation rule* (of an SADT diagram) is a logical expression stating the relation among input, control, and output arrows.[4] The *activation rule* states the conditions (presence or absence of data) on the input, control, and output arrows such that presence of data on the output arrows will occur. *Activation rules* transform SADT diagrams into finite-state machine descriptions.

An *activation* is a path traced through a *model* from its external input and control arrows to its external output arrows, according to the *activation rules* controlling each diagram of the *model*.

A *scenario* is a set of values applied to the external arrows of a *model* before tracing an *activation* of the *model*.

A *concept of operation* of a *system* is a *model* containing a *functional model* of that *system* along with *functional models* of *systems* it interacts with. A *concept of operations* specifies the set of all valid *activations* of a *system*.

An *example of a concept of operations* is an *activation* of a *concept of operations* for some *scenario*.

A *designer* is a *vantage point* of a person who choses at each hierarchical level of detail of a *model* a partitioning that results in the least coupled pieces.

A *design model* of a *system* is a *functional model* of that *system* repartitioned from the *view* of a *designer*. The *functional model* maps into, not onto the *design model*, since additional information about the *system* may be incorporated in the *design model*.

A *specification* is a set of *activation rules* applied to the *design model* to constrain its set of possible *activations* to be in the bounds described in the *functional model*. Some of these constraints are necessary to control the effects of the additional information the *design model* contains.

THE CHAIN OF DOCUMENTS

The chain consists of the following documents:

Concept of operations, functional model, design model, specification, as defined in the previous section.

The *concept of operations* describes how a new system will be used, together with the existing resources and weapons, to get a net increase in the overall effectiveness of the entire set of resources. Considered by itself, only system performance can be measured. Considered in its context, system effectiveness can be measured. The concept of operations provides a purpose, view, and context for a functional model. It binds the document chain to the operational context.

The *functional model* describes how the system must behave and what it must be able to do for its user. The user's requirements for speed, accuracy, size, etc. are documented in the functional model. Later this document will be used by the implementer to understand the user's "utility function" so implementation trade-offs can be made to satisfy the user (rather than vague ideas of efficiency). The functional model binds the problem description to the operational context.

The *design model* describes the modular structure (logical structure) of the system to be built. As defined above, it is the "simplest" description of the system in terms of constantines "structured design." [5],[6],[7] The functions in this model are allocated to hardware or software according to the trades made by the system engineers. The structure constrains, but does not specify, the algorithms to be used. This model binds problem structure to solution structure.

The *specification* chooses the logic or algorithms of the system to get the behavior required by the functional model. The specification binds solution behavior to problem behavior.

The design model and specification do not include any properties of the hardware or software used to implment the system. They do constrain the implementer to produce a system of a certain structure and behavior, and guide tradeoffs among alternatives with a utility function from the functional model.

	PROBLEM DESCRI-PTION	SOLUTION DESCRI-PTION	IMPLEMENTATION DESCRIPTION
Required Capability	why		
Concept of Operation	what	why	
Functional Model	how	what	why
Design Model		how	what
Specification			how

LINKS IN THE CHAIN

For the documents to form a chain, they must be linked. These links are made in the SADT models forming the chain. The SADT syntax provides a way of formally linking the models.

The concept of operations contains the functional model embedded within it. The functional model is, in a sense, a named subset of the concept of operations, drawn from another view. The design model is linked to the functional model using the SADT mechanism syntax. Each function in the functional model contains a mechanism "call" reference to the portion of the design model that realizes that function. The design model can, if desired, be annotated with SADT support arrows which relate portions of the design back to the functions in the functional model. Thus, requirements traceability back and forth from functional model to design is maintained.

The specification is an annotation of the design model. Thus no cracks between the models that accumulate "noise" are permitted.

The argument to this point has relied somewhat on the properties of SADT models. However, the concept is more general than that. It should work for a modeling technique with the following properties:

1. Describes modular structure, hierarchically.

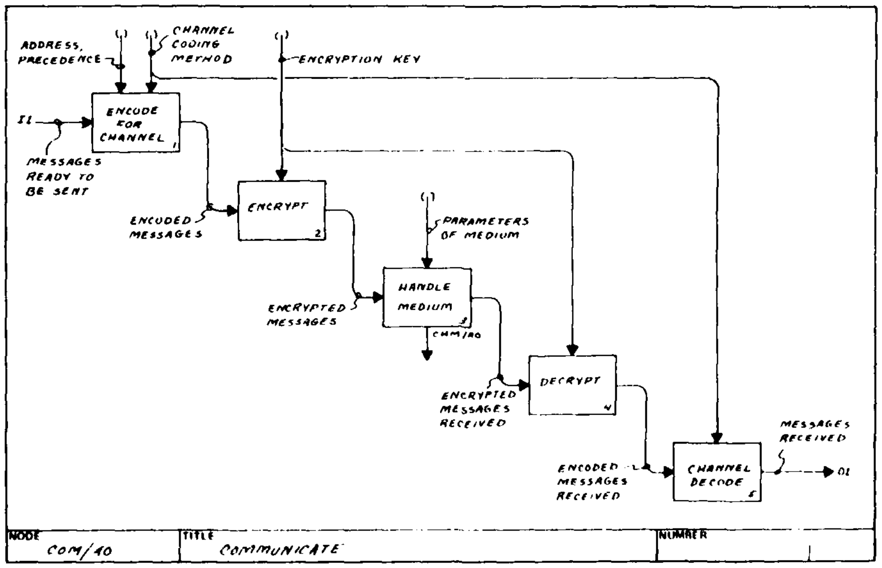
2. Provides for traceability from model to model.

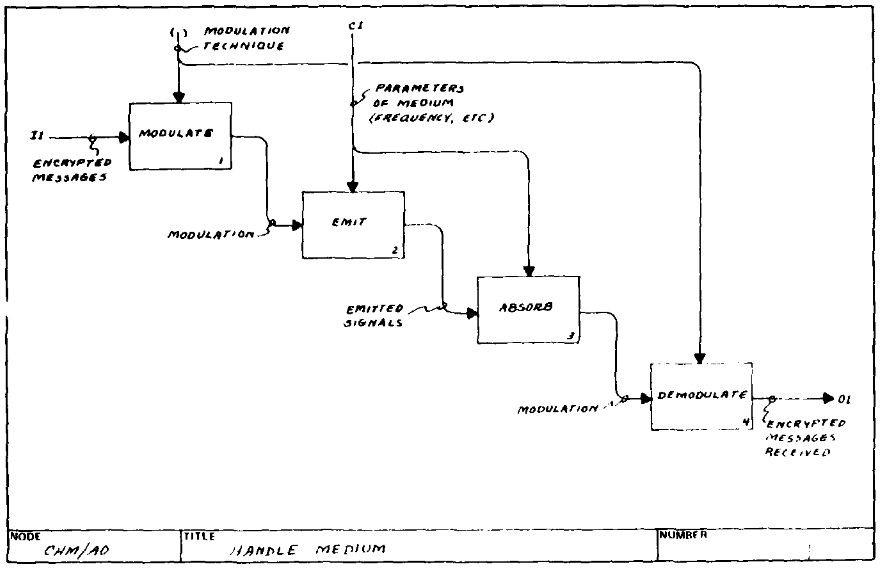
3. Contains a way of specifying activations in terms of output value "events" under some input value "conditions."

SADT has been used because it is a natural language for describing system structure hierarchically. At each level, the pieces and their relations (dependencies) are expressed. The mechanism syntax allows traceability between models. Activation rules directly transform SADT models into finite-state machine descriptions. And, the language is a simple, graphic one.

As an illustration of how the mechanism notation can connect one model to the next, consider the two SADT diagrams "communicate" and "handle medium." Each diagram is the first level decomposition of a different model.

Box 3 on "communicate" has a downward pointing mechanism arrow that calls the A0 diagram of model CHM. Thus the "Handle Medium" diagram realizes a box in another model.





ADEQUACY OF THE MODELS

To evaluate the adequacy of a model in the chain, each model is associated with a development test, as follows:

MODEL	TEST	MEANING
concept of operations	exercise	Is it useful?
functional	acceptance	Does it work?
design	integration	Do the pieces go together?
specification	verification	Are the pieces correct?

A model is adequate if it can be used to develop a plan for its corresponding test. For users, the concept of operations is sufficient if it can be used to outline an exercise that uses the new system. More requirements are forced out early in the process by consideration of details of the test.

This criterion also suggests what should be excluded from models. Information beyond that necessary to construct the test should be viewed with suspicion; it may be merely extraneous, it may belong in a different model, and its inclusion at this level may overconstrain the next level's model. Not everything you know about a system needs to be in the next document you write about the system.

A MODEL OF THE CHAIN OF MODELS

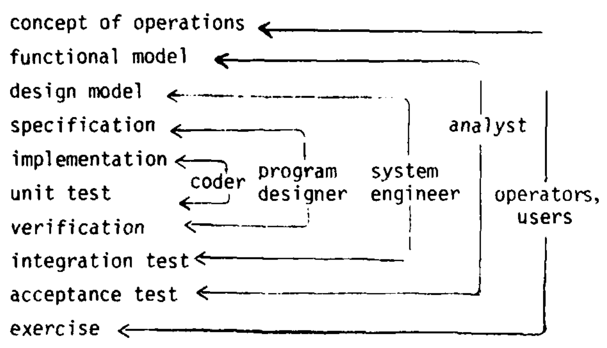
The SADT diagrams of this section present a model of a Naval system development from the viewpoint of a program manager. The diagrams are annotated to show where the documents of the chain can be used to help satisfy the requirements of DSARC reviews that lead to formal milestone decisions.

Each document of the chain appears at a different level of detail in the model. The concept of operations appears at the first level on the A0 diagram. The functional model appears at the second level on the A3 diagram, and so on. All of these diagrams (A0, A3, A33) that describe a document in the chain have the same form; each is related to its corresponding test. Diagram A1 corresponds to the concept formulation stage in a major system acquisition.

There are three reasons for including this model: it gives a sample of an SADT model for those unfamiliar with the method, it provides a scheme for delegation of design authority, and it provides a framework to discuss the order in which the documents are produced in a real development.

The method of delegation of design authority embodied in the model was inspired by Cowen [8]. At the operational level, "Build," Box 3, is delegated, while authority for the other functions is retained. Box 3 on A3 and A33 is similarly delegated. Each level is responsible for a model in the chain. Each delegation is accompanied by a constraint to similiarly delegate and constrain.

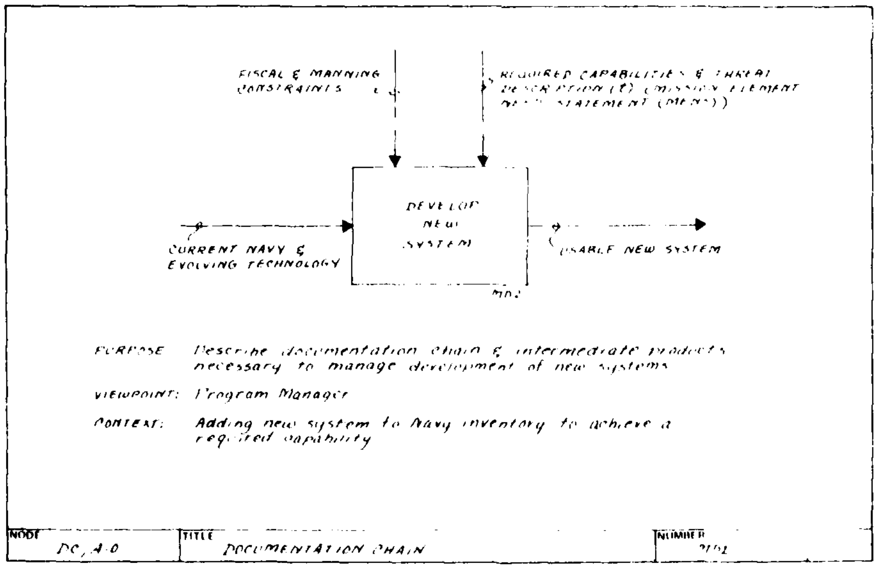
Responsibilities work out like this:

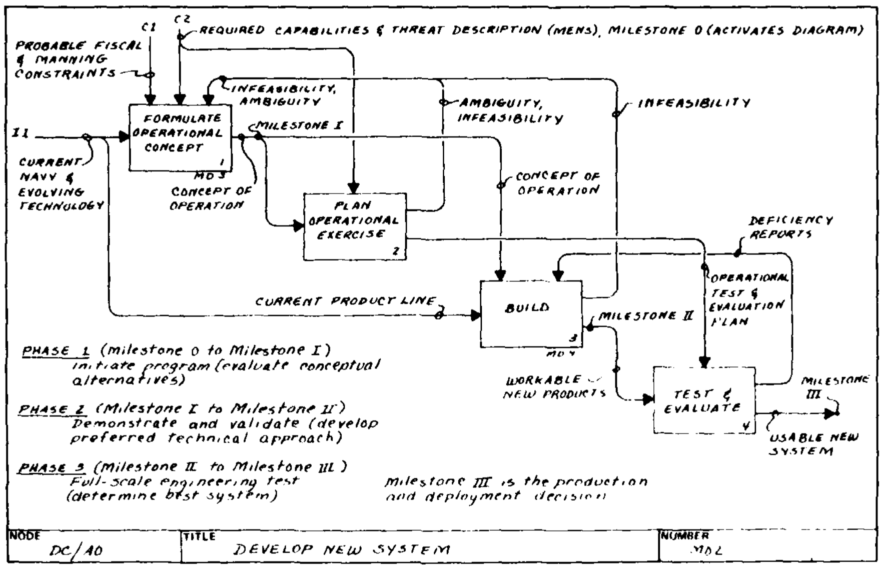


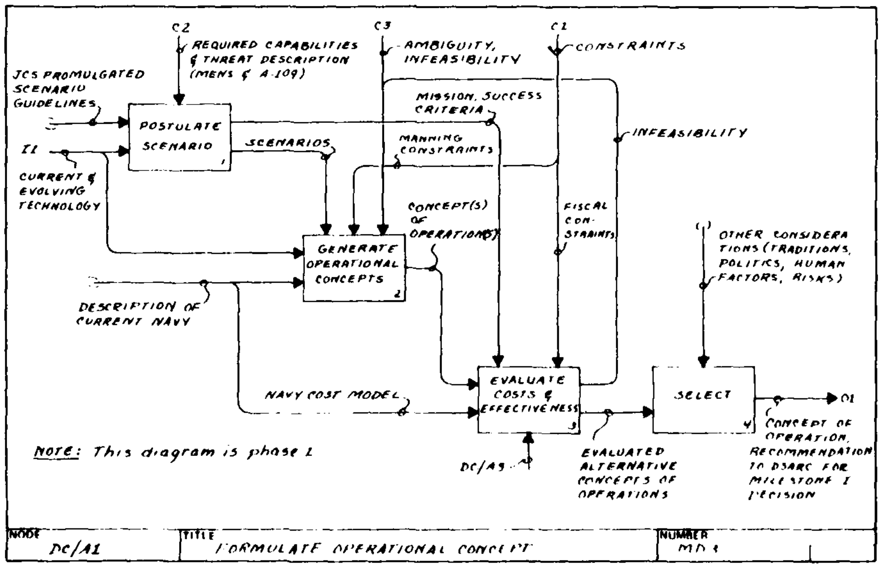
The model shows the dependencies of a set of activities and the documents they produce. It constrains, but does not specify, the exact order in which the development activities are carried out. For example, there are feedback loops on diagram A0 that may cause activities to be repeated, depending on how feasibilities work out. Opportunities for subtle activation paths abound. On Diagram A1, Box 3 has a mechanism support arrow that indicates Diagram A3 (Build) may be invoked in the course of evaluating costs and effectiveness. This could arise if it becomes necessary to build a feasibility demonstration of some risky part of the system to fully evaluate an alternative.

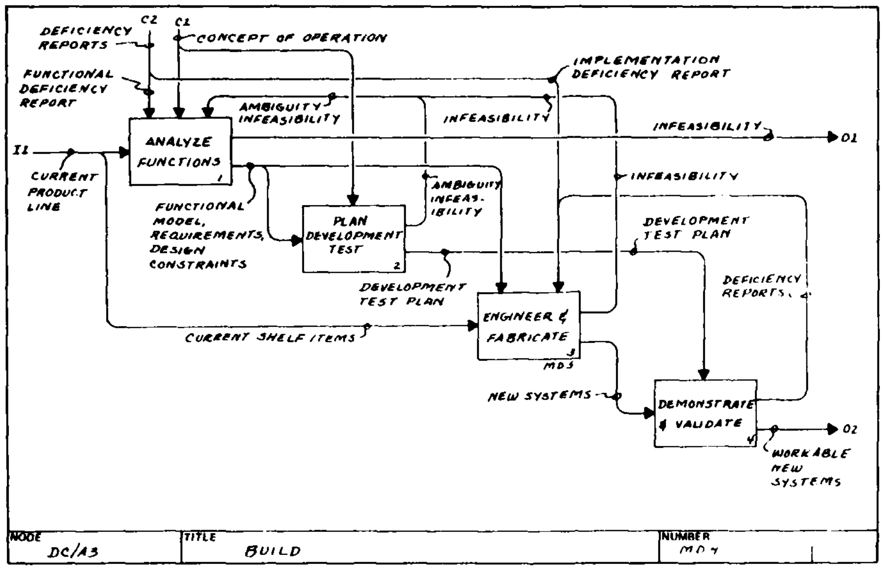
The model allows many alternate activation paths and dynamic behavior patterns in system development; however the documents produced at the end are constrained to relate to each other in a particular way.

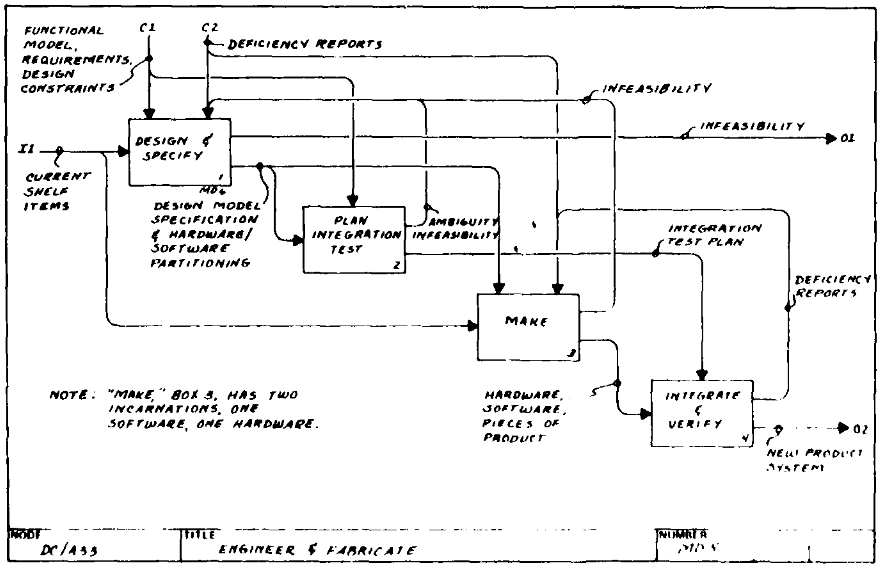
In practice, an activation of this model may begin with a functional model rather than a concept of operations. Technologists may propose a functional model of a new kind of system to find a use for a new technology. A concept of operations would be created for it, along with concepts of operations for competing alternatives. A revision of the external details of the functional model would likely result, and a new functional model would then be created in depth.

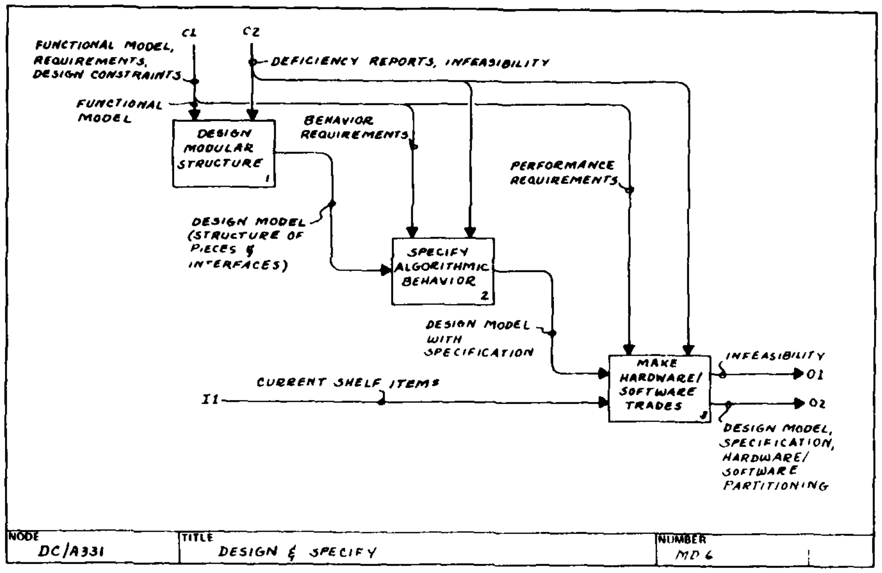

FISCAL & MANNING CONSTRAINTS
REQUIRED CAPABILITIES & THREAT DESCRIPTION (MISSION ELEMENT NEED STATEMENT (MENS))
DEVELOP NEW SYSTEM
CURRENT NAVY & EVOLVING TECHNOLOGY
USABLE NEW SYSTEM
PURPOSE Describe documentation chain & intermediate products necessary to manage development of new systems
VIEWPOINT: Program Manager
CONTEXT: Adding new system to Navy inventory to achieve a required capability
NODE DC/A-0
TITLE DOCUMENTATION CHAIN
NUMBER


C1
C2
REQUIRED CAPABILITIES & THREAT DESCRIPTION (MENS), MILESTONE 0 (ACTIVATES DIAGRAM)
PROBABLE FISCAL & MANNING CONSTRAINTS
INFEASIBILITY, AMBIGUITY
AMBIGUITY, INFEASIBILITY
INFEASIBILITY
I1
FORMULATE OPERATIONAL CONCEPT 1
MD 3
MILESTONE I
CURRENT NAVY & EVOLVING TECHNOLOGY
CONCEPT OF OPERATION
CONCEPT OF OPERATION
PLAN OPERATIONAL EXERCISE 2
DEFICIENCY REPORTS
OPERATIONAL TEST & EVALUATION PLAN
CURRENT PRODUCT LINE
BUILD 3
MD 4
MILESTONE II
WORKABLE NEW PRODUCTS
TEST & EVALUATE 4
MILESTONE III
USABLE NEW SYSTEM
PHASE 1 (milestone 0 to Milestone I)
initiate program (evaluate conceptual alternatives)
PHASE 2 (Milestone I to Milestone II)
Demonstrate and validate (develop preferred technical approach)
PHASE 3 (Milestone II to Milestone III)
Full-scale engineering test (determine best system)
Milestone III is the production and deployment decision
NODE DC/A0
TITLE DEVELOP NEW SYSTEM
NUMBER MD2


C2
REQUIRED CAPABILITIES & THREAT DESCRIPTION (MENS & A-109)
C3
AMBIGUITY, INFEASIBILITY
C1
CONSTRAINTS
JCS PROMULGATED SCENARIO GUIDELINES
I1
CURRENT & EVOLVING TECHNOLOGY
POSTULATE SCENARIO 1
SCENARIOS
MISSION, SUCCESS CRITERIA
MANNING CONSTRAINTS
INFEASIBILITY
CONCEPT(S) OF OPERATION(S)
FISCAL CON-STRAINTS
GENERATE OPERATIONAL CONCEPTS 2
DESCRIPTION OF CURRENT NAVY
OTHER CONSIDERATIONS (TRADITIONS, POLITICS, HUMAN FACTORS, RISKS)
NAVY COST MODEL
EVALUATE COSTS & EFFECTIVENESS 3
SELECT 4
O1
DC/A9
EVALUATED ALTERNATIVE CONCEPTS OF OPERATIONS
CONCEPT OF OPERATION, RECOMMENDATION TO DSARC FOR MILESTONE I DECISION
NOTE: This diagram is phase 1
NODE DC/A1
TITLE FORMULATE OPERATIONAL CONCEPT
NUMBER MD 3


C2
C1
DEFICIENCY REPORTS
CONCEPT OF OPERATION
FUNCTIONAL DEFICIENCY REPORT
IMPLEMENTATION DEFICIENCY REPORT
AMBIGUITY INFEASIBILITY
INFEASIBILITY
INFEASIBILITY
O1
I1
ANALYZE FUNCTIONS 1
CURRENT PRODUCT LINE
INFEASIBILITY
AMBIGUITY INFEAS-IBILITY
FUNCTIONAL MODEL, REQUIREMENTS, DESIGN CONSTRAINTS
PLAN DEVELOPMENT TEST 2
DEVELOPMENT TEST PLAN
DEVELOPMENT TEST PLAN
DEFICIENCY REPORTS
ENGINEER & FABRICATE 3
MD5
CURRENT SHELF ITEMS
NEW SYSTEMS
DEMONSTRATE & VALIDATE 4
O2
WORKABLE NEW SYSTEMS
NODE
DC/A3
TITLE
BUILD
NUMBER
MD4


FUNCTIONAL MODEL, REQUIREMENTS, DESIGN CONSTRAINTS
C1
C2
DEFICIENCY REPORTS
INFEASIBILITY
I1
DESIGN & SPECIFY 1
MD6
INFEASIBILITY
O1
CURRENT SHELF ITEMS
DESIGN MODEL SPECIFICATION & HARDWARE/ SOFTWARE PARTITIONING
PLAN INTEGRATION TEST 2
AMBIGUITY INFEASIBILITY
INTEGRATION TEST PLAN
DEFICIENCY REPORTS
MAKE 3
NOTE: "MAKE," BOX 3, HAS TWO INCARNATIONS, ONE SOFTWARE, ONE HARDWARE.
HARDWARE, SOFTWARE, PIECES OF PRODUCT
INTEGRATE & VERIFY 4
O2
NEW PRODUCT SYSTEM
NODE
DC/A33
TITLE
ENGINEER & FABRICATE
NUMBER
MD5


C1
C2
FUNCTIONAL MODEL, REQUIREMENTS, DESIGN CONSTRAINTS
DEFICIENCY REPORTS, INFEASIBILITY
FUNCTIONAL MODEL
DESIGN MODULAR STRUCTURE 1
BEHAVIOR REQUIREMENTS
PERFORMANCE REQUIREMENTS
DESIGN MODEL (STRUCTURE OF PIECES & INTERFACES)
SPECIFY ALGORITHMIC BEHAVIOR 2
DESIGN MODEL WITH SPECIFICATION
CURRENT SHELF ITEMS
I1
MAKE HARDWARE/ SOFTWARE TRADES 3
INFEASIBILITY
O1
O2
DESIGN MODEL, SPECIFICATION, HARDWARE/ SOFTWARE PARTITIONING
NODE DC/A331
TITLE DESIGN & SPECIFY
NUMBER MD 6

CONCLUSIONS

By establishing a rigorously linked chain of documents connecting requirements in their operational context to a specification of what the implementer must build, the accumulation of noise in the process is prevented.

By introducing a test for the adequacy of each model, more requirements are forced out early in the process, and attention is focused earlier on feasibility issues.

This method has not been tried. However, SADT has been used to construct a Concept of Operations. A functional model has been built and transformed into a design model [9]. As yet, no design model has been completely turned into a specification using activation rules; for various reasons other specification languages (e.g. programming languages), have been used. The author intends to try the technique on a software development in the near future.

REFERENCES

1. Ross, D.T., Quality Starts with Requirements Definition. P.G. Hibbard and S.A. Schoman Eds. North Holland Pub. Co., 1978, pp. 397-406.

2. Ross, D.T. and Schoman, K.E., Structured Analysis for Requirements Definition. IEEE Trans. on Software Engineering, 3. 1, 1977, pp. 6-15.

3. Ross, D.T., Structured Analysis: A Language for Communicating Ideas. IEEE Trans. on Software Engineering, 3. 1, 1977, pp. 16-34.

4. Dickover, M.E., McGowan, C.L., Ross, D.T., Software Design Using SADT, Infotech State of the Art Report: Structured Design, Maidenhead, Infotech International Ltd, 1978.

5. Yourdon, E. and Constantine, L.L.: Structured Design. Yourdon, Inc., New York, 1975.

6. Dickover, M.E., Principles of Coupling and Cohesion for Use in the Practice of SADT. SofTech, Inc., Waltham, Mass., Technical Publication #039, 1976.

7. Alexander, C., Notes on the Synthesis of Form, Cambridge, Harvard University Press, 1964.

8. Cowan, G. Jr., A General Structure for Resource Management in a Computer Network. PhD. Thesis, Madison: Univ. of Wisconsin, 1975.

9. Dickover, M.E. and Small, A., Analyzing and Designing an EW Reprogramming System, Proceedings of the 41st Symposium of the Military Operations Research Society, July 1978.

ACKNOWLEDGEMENT

The ideas in this paper were developed during work for Dr. Michael Melich of the Naval Research Laboratory. Many hours of discussion with him led to these ideas, and his encouragement was responsible for their development. The Structured Analysis and Design Technique (SADT - a trademark of SofTech, Inc.) was devised by Douglas T. Ross.

A STEP TOWARDS THE OBSOLESCENCE OF PROGRAMMING

Harvey S. Koch

University of Rochester
Graduate School of Management
Rochester, New York 14627

Abstract

One of the most difficult aspects of software development is transforming the systems analyst's specifications into process specifications that can be understood and carried out by a computer. We describe a method in this paper that can be used to reduce the complexity of the programmer's task and to ensure a higher degree of correlation between the systems analyst's specifications and the program produced for the application.

Currently, the programmer has the task of interpreting and translating the user's requirements into specifications that can be understood by the computer. As a result, a common phenomenon associated with each implementation of a system is that the user's requirements have not been satisfied. To begin to solve this problem, we propose that the complexity of the programming should be reduced. This will reduce the number of errors committed by programmers and will allow what the systems analyst specifies to be more accurate.

Ideally, we are striving towards the elimination of programming in the form that it currently exists. In this paper, we describe one method that we are trying to apply as a means to this end. The basic ideas of our methodology have been taken from [1]. More research is needed to assess its potential.

We will describe our methodology in terms of an example. Then, we will summarize the characteristics of our approach.

Our example is a bank transaction application. Specifically, the processing of a universal form for deposit, withdrawal and transfer of funds between accounts. The input fields are:

1. date
2. bank office
3. teller
4. type of transaction
5. account number(s)

The systems analyst usually specifies the operations to be performed on each data field. An example of the specifications are:

1. Date
   a. check whether date is current
   b. calculate interest if not already done for previous period(s)
2. Bank Office
   a. check whether valid
3. Teller
   a. check whether teller number is valid for that office
4. Type of Transaction
   a. deposit into account1
   b. withdrawal from account1 - check whether balance > 0
   c. transfer from account1 to account2 if balance in account1 > 0
5. Account1
   a. check whether still active
6. Account2
   a. check whether still active.

An important characteristic of the systems analyst's specifications is that they are usually non-procedural and are centered around the data. The programmer, however, has the duty of taking these non-procedural statements and transforming them into a program - i.e., something that is procedural and process-oriented. There is much decision making that the programmer must do to create a process for the computer. Specifically, he must decide:

1. In what order the fields should be processed.
2. What are the controls of instruction execution.

3. What must be synchronized and what are the conditions for synchronization if parallel processing is used.

Our methodology allows the programmer to create independent portions of a program and uses a translator to produce a program which interrelates these independent program protions. For each field of the input data, using the systems analyst's specifications, we have a graphic representation:

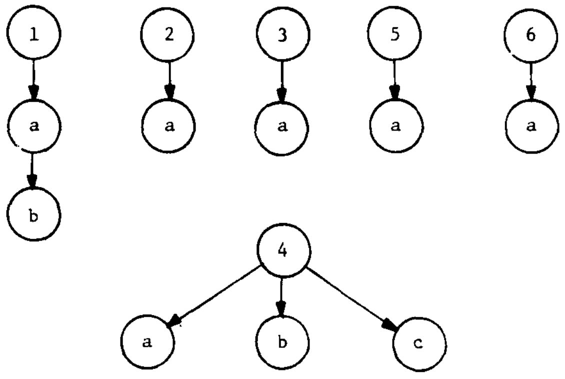


FIGURE 1

Independent Program Portions

A node with a number represents the fact that the succeeding instructions should be executed to process that field. A node with a letter means that the instruction(s) associated with that node should be executed. For instance, the left-most graph represents the fact that two operations ("a" - check whether date is current and "b" - calculate interest if not already done for previous period(s)) should be executed to process the "date" field. First, the instruction(s) for "a" should be executed and then the instruction(s) for "b". Since only one of "a", "b", or "c" is executed for field 4, these three nodes are not represented in sequential fashion.

What we have represented in Figure 1 are data flow graphs for each input data field. Each node of the graph represents an instruction or a sequence of instructions. All instructions that operate on the same field are chained together.

At this point, the specification step has been completed. We still have not specified, though, what fields can be processed in parallel and what synchronization events are needed. In existing design and implementation methodologies these are defined in the design stage or as late as the time of coding.

The graph below represents, in terms of data flow, the operations necessary to check data integrity, to process the data plus which instructions can be processed in parallel and which operations must be synchronized.

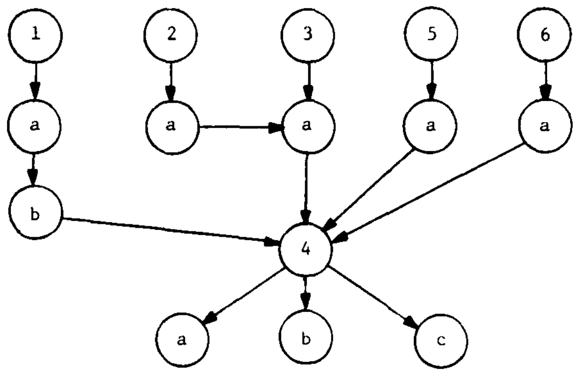


FIGURE 2

Relationship Between the Program Portions

The operations in fields 1, 2, 5, and 6 can begin to be executed in parallel since the operations on these fields begin with intra-field relationships. Field 3 begins with an integrity check that involves an inter-field relationship. Hence, 2a must be completed before 3a begins. Operations 4a, 4b, and 4c must wait for all other processing and integrity checks to have been completed.

A translator can produce a program equivalent to the graph in Figure 2 given the instructions represented in Figure 1. Instructions for each data field can be executed in parallel until another data field is referenced. When an instruction references another data field, all previous instructions that reference that data field must have finished execution.

In summary, we need to bridge the gap between systems analysis and programming. We must look for methods that foster (semi) automatic translation between the systems analyst's specifications and the program code.

Our methodology assumes that the modules of the system have been identified in the design stage and that the data items to be operated on in each module are known. For each data item, the systems analyst then specifies what are the operations to be performed. The programmer then writes independent program portions for each data item. A translator is then used to produce the program which interrelates the program portions. The end result is:

1. closer interaction between the systems analyst and the programmer since the systems analyst writes his specifications after the design phase
2. reduction of decision-making responsibilities for programmers; specifically he does not decide on the sequencing of instructions on different data items nor on any synchronization conditions.

3. the design of each module is based upon its data items.

Our methodology can alleviate some of the inconsistencies between the user's requirements and the programmer's end product. Programming is still necessary but it has been reduced in its complexity. We anticipate future research by us and others to further reduce the complexity of programming and to change it from a process-oriented activity. We hope that in the near future, the process-oriented method of programming will become obsolete.

References

1. B. Gavish and H. Koch, "An Extensible Architecture for Data Flow Processing," Proceedings of the Fourth Workshop on Computer Architecture for Non-Numeric Processing, August 1978.

A CONTINGENCY THEORY TO SELECT AN INFORMATION REQUIREMENTS DETERMINATION METHODOLOGY

J. David Naumann
Gordon B. Davis

University of Minnesota
College of Business Administration
Minneapolis, Minnesota 55455

The system development life cycle is the central concept in currently-used methods of managing and controlling the determination of information requirements and designing and implementing processing systems to meet those requirements. When organizations specify the use of formal life-cycle-based methods for all application developments, the results are mixed. The method may be helpful, but in many cases, it may be detrimental.

A single life-cycle method is not appropriate for all cases because applications differ in the certainty with which requirements can be established. This paper describes the contingencies which define the uncertainties in the determination of information requirements and describes alternative strategies for information requirements determination given different levels of uncertainty about information requirements. The methods are no cycle, linear life cycle, recursive life cycle, and prototype.

Introduction

Formal life cycle development methodologies are not consistently reflected in practice; concepts such as throw-away code, prototype systems, and recursive life cycles are receiving increasing attention in industry. The profession has developed a paradox: the life cycle procedures when carefully followed provide a high degree of assurance of project success for many systems, but they may be prohibitively expensive and unwieldly for many others. This suggests that there may be different methods for different projects.

A contingency theory is a theory which identifies alternative actions and presents factors to use in selecting the optimal alternative. For example, McFarlan[1] proposes a contingency theory for development project management. He identifies project size, degree of structuredness, and degree of company-relative technology as factors which determine the best planning and control tool for a project.

Uncertainty has been identified by Galbraith[2] as a major factor in determining the optimal organization structure. The difference between the amount of information necessary to perform a task and the amount of information possessed is a measure of task uncertainty; organizations respond by choosing from a set of four organizational strategies to deal with the level of uncertainty.

The information requirements determination portion of information systems development can be viewed as a problem in uncertainty; the formal life cycle methodology is a response to this uncertainty. Its intent is to make sure that the organization does enough information processing (with regard to requirements) that uncertainty is reduced to acceptable levels. However, since the information requirements uncertainty differs widely for different applications, a single formal life cycle approach is incapable of addressing the full range of information systems development projects.

Contingencies

In the determination of information requirements for an information system application uncertainty refers to knowledge of the "real" information needs. Development contingencies which determine information requirements uncertainty are project size, degree of structuredness, user-task comprehension, and developer-task proficiency. A systems development project has some combination of these attributes (and perhaps other as yet unspecified contingencies). The combination of contingencies determines the choice of development methodology.

Project Size

The project size contingency has three key characteristics: duration, number of people involved, and total dollar amount. These characteristics are usually, but not necessarily, collinear. That is, a high cost project usually requires many people over an extended time period. Project size is not a good measure of the value of a systems development project, but it is correlated with the degree of uncertainty of the results of the development process.

Degree of Structuredness

One dimension of the Gorry and Scott Morton (1971) framework for information systems is that of the relative structuredness of the decisions to be supported by an information system.[3] For information systems information requirements determination, a high degree of structuredness means that a general model exists which needs only to be applied to the given organizational setting. A low degree of structuredness means that there is no routine procedure for dealing with the problem, there is ambiguity in the problem definition and uncertainty as to the criterion for evaluating solutions. Uncertainty about the decision to be supported is an important factor in uncertainty about the outcome of the systems development process.

User Task Comprehension

Related to but distinct from structuredness is the comprehension that the user or users have of the task to be performed by the information system. User task comprehension affects the strategy and development project success in much the same way as degree of structuredness. If the users have a low degree of understanding of the task for which the system is intended, whether or not a general model of a problem exists, less is certain about the information requirements (and the users' acceptance of the results of the development process).

Developer Task Proficiency

Developer task proficiency is a measure of the specific training and experience brought to the project by the development staff: project manager, liaison staff, systems analysts, systems designers, programmers, etc. It is not a measure of ability or potential: rather it is a measure of directly applicable experience. This contingency indicates the degree of certainty with which the developer will be able to obtain and document the requirements (and also proceed with the remainder of the development process).

Uncertainty-Reducing Strategies

The response to uncertainty produced by characteristics of a systems development task, the using organization, and the developer organization (i.e., the contingencies) has frequently been unidimentional. Under the traditional life cycle approach, formal procedures, reviews, committees, check points, etc. are used for all projects.[4] There has been no recognition of the degree of uncertainty from the contingencies. An alternative approach is to:

1. Identify the contingencies and determine the uncertainty,
2. Select an information requirements determination method suitable for the level of uncertainty.

Figure 1 shows the relationship of uncertainty-reducing strategies to level of uncertainty, and suggests methodologies which are actualizations of these strategies. The strategies are: accept information requirements as specified, linear discovery, recursive discovery, or experimental discovery of information requirements. Each has an associated method.

Accept as Specified

If information requirements are known and agreed upon, then the proper strategy is to accept the user's statement of need as adequate specification for implementation. The method is therefore to have no information requirements cycle. Examples are file conversions, reports from existing files or databases and small single-user models. These examples have in common: small size, high degree of structure, users who understand what the systems are to do and how the implementation will function, and developers who are experienced in this kind of task. Explicit recognition of the need for the "accept as specified" *strategy will* lead to greater responsiveness and an increase in development organization efficiency.

Linear Discovery

If information requirements can be determined through a straight-forward process of interviewing, fact gathering, and documentation, the proper strategy is to proceed step-by-step to system specification. The method is therefore a linear application of the life cycle. Examples are transaction level systems, single function accounting systems such as accounts receivable or payable, and minor modifications to existing information systems.

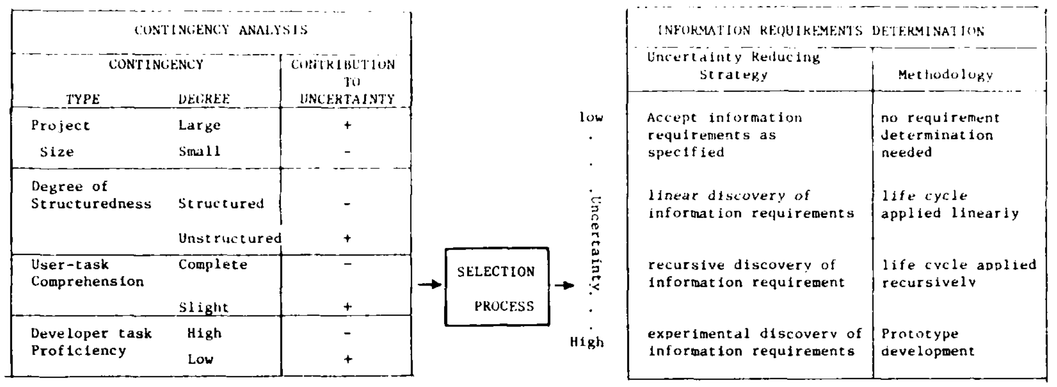
The information requirements for large systems which are highly structured and where user-task comprehension and developer-task proficiency are high may be effectively determined by the linear discovery process. However, information requirements for a relatively small system may not be determinable by this method if the decisions to be supported are relatively unstructured, or if the user does not comprehend the task, or if the developers have not previously produced such a system. Linear application of the life-cycle model is an effective strategy under the appropriate combination of contingencies.

Recursive Discovery

The linear discovery strategy may not produce correct or complete or acceptable specifications of information system requirements. The traditional life-cycle approach extends to recursion for such systems. One or more discovery tasks are iterated until a complete, consistent specification is determined and accepted. Examples are large, multiple-user systems, systems which are new to the user or developer organization, and systems which support the relatively unstructured decisions of tactical and strategic management. This approach assumes that a correct specification of requirements can be made given sufficient time and effort. Where the contingencies indicate that is a valid assumption, the recursive discovery strategy is appropriate and effective.

Experimental Discovery

A high level of uncertainty may be indicated by a combination of the contingencies. Repeated iterations of discovery may not successfully produce adequate specifications of information requirements in such cases. The life cycle method, whether linear or recursive, is inappropriate when uncertainty is high. The strategy of experimental discovery as realized in the prototype design method, reduces uncertainty by producing successive



CONTINGENCY ANALYSIS		
CONTINGENCY		CONTRIBUTION TO UNCERTAINTY
TYPE	DEGREE	
Project Size	Large	+
	Small	-
Degree of Structuredness	Structured	-
	Unstructured	+
User-task Comprehension	Complete	-
	Slight	+
Developer task Proficiency	High	-
	Low	+

INFORMATION REQUIREMENTS DETERMINATION	
Uncertainty Reducing Strategy	Methodology
Accept information requirements as specified	no requirement determination needed
linear discovery of information requirements	*life cycle* applied linearly
recursive discovery of information requirement	life cycle applied recursively
experimental discovery of information requirements	Prototype development

FIGURE 1

SYSTEMS DEVELOPMENT METHODOLOGY SELECTION CONTINGENCY MODEL

approximations. Users and developers can easily see what is wrong with an implementation even though they are unable to completely specify its information requirements.[5] Examples of appropriate applications of the experimental discovery approach are decision support systems for upper management, interactive forecasting models, and small (or large) systems to be implemented for many different users. Conscious selection of the experimental discovery strategy may be the only effective approach to information requirements determination when the level of uncertainty is high.

Conclusion

A range of information requirements determination strategies is needed. Such strategies match the level of uncertainty about the system specification which is to result. The appropriate strategy is determined by the extant contingencies.

Research is needed in two major areas of the contingency theory: more precise operational definitions of the contingencies, and identification and definition of other factors which effect the level of uncertainty, should be demonstrated. The range of discovery strategies and corresponding information requirements determination methods may be extended. The results of such research will lead to improved selection of methodology, and more effective systems development.

References

1. F. W. McFarlan, "Effective EDP Project Management," in Managing the Data Resource Function (R. Nolan, Ed.), West Publishing Company, St. Paul, 1974.

2. J. Galbraith, Designing Complex Organizations, Addison-Wesley, Reading, MA, 1973.

3. G. A. Gorry and M. S. Scott Morton, "A Framework for Management Information Systems," Sloan Management Review, Volume 13, Number 1, (Fall 1971), pp. 55-70.

4. G. B. Davis, Management Information Systems: Conceptual Foundations, Structures, and Development, McGraw-Hill, New York, 1974.

5. C. Alexander, Notes on the Synthesis of Form, Harvard University Press, Cambridge, MA, 1964.

A LIFE-CYCLE MODEL BASED ON SYSTEM STRUCTURE

F.N. PARR

Department of Computing and Control
Imperial College, London

Abstract

A new model for the software development process is presented in which the rate of progress is determined by the extent to which the system under development can be decomposed into modules which permit independent development. The pattern of resource consumption of a project over its life-cycle can be derived from this model. The results are compared with previous predictions of a Rayleigh curve pattern.

Introduction

PUTNAM and NORDEN (1),(2) have proposed a life cycle model for software development effort which predicts that the rate of resource consumption on a project should have a Weibull distribution in time. In other words if W(t) is the proportion of the total project completed by time t then the rate of progress is give by

$$\frac{dW}{dt} = p'(t) \cdot e^{-p(t)}$$

and

$$\int_0^t \frac{dW}{dt}.dt = 1 - e^{-\int^t p(u)du} \qquad (1).$$

[illegible] behind this prediction [illegible] follows. The objective [illegible]rresponds to some larg[illegible] [illegible]roblems to be solved [illegible] design decisions to [illegible] making these [illegible]ther actually [illegible]ware. The rate [illegible] is assumed [illegible] to the [illegible] at time [illegible] available [illegible] which we [illegible]

$$\frac{dW}{dt} = p'(t) \, . \, (1 - W(t)) \qquad (2)$$

from which equation (1) can be derived.

The function p'(t) defined by this expression is taken to be a "learning curve" which describes how the development team becomes increasingly skillful in making decisions or solving problems associated with this particular project. Any function p'(t) can be plugged into the model and will generate a predicted life-cycle pattern belonging to the Weibull family. However a reasonable fit with empirical data from actual software development projects can be obtained by choosing a linear learning function $p'(t) = a.t$. The life-cycle pattern resulting from this choice is the Rayleigh function

$$\frac{dW}{dt} = a.t.e^{-at^2/2}$$

The general shape of the Rayleigh curve matches a very simple intuitive picture of the life-cycle of software. There is an initial design phase during which the rate of resource consumption increases steadily, a central peak of effort corresponding to principal implementation activity and a maintenance phase during which the work rate decays back to zero. These phases of development may be identified with the different assumptions which made up the model of the development process. The gradual decay of work rate during maintenance is basically the result of the problem space becoming exhausted. Since it was assumed that the goal of the software project was some task which could be characterised by a fixed finite pool of problems to be solved, then as the work progresses this pool must eventually empty. This must cause some form of exponential slowing down of maintenance effort. However, the Rayleigh model's explanation of the initially growing work rate in terms of increasing skill level being available is open to several criticisms.

Firstly, a skill level which just grows linearly may be rather unrealistic. Often the staff joining a development project will have worked on closely related projects before. There is little reason to suppose that the skill of these people improves. It is also the case that programmers remain with a system only for a portion of its life-cycle. In that case there is no reason for those who join late on to be more skillful than those who started. Taken to extremes, when a system is in the final maintenance stage of its life and probably has low priority it would be most unlikely to attract the most expert programmers as the theory suggests.

Now it can be argued that linearity is only a first approximation and that these objections can be answered by choosing a more realistic form for the learning curve p'(t). However, the model as presented gives little guidance as to what functions for p'(t) would be acceptable. The actual property being measured by p'(t) is not sufficiently precisely defined to allow an empirical approach. Ideally one would like to measure the skill level on some actual projects and fit a curve to it. In some sense the Rayleigh function model merely transforms the problem rather than solving it; the difficulty of predicting the pattern of resource consumption of a project is exchanged for the difficulty of estimating how the skill level changes.

A final objection is that the Rayleigh model fails to distinguish between innate constraints on the process of writing software and management's economically constrained hiring and methodology decisions. The trouble is that the level of skill available on a project depends jointly on how long skilled personnel have been assigned to it and also on the tools and design methods which they use. A model based only on skill level therefore presents a picture of software development as being an unalterable process. A more powerful theory would show how the pattern of development may be altered as a result of the introduction of new software engineering methods.

This paper presents an alternative life-cycle model for software development which is based much more closely on the nature of the development process. The notion of skill level is replaced with the idea that strong modularity in a program is what governs the rate of progress since it allows work by several teams to proceed in parallel. This concept clarifies the role of project management's apparently free choice to hire staff while at the same time suggesting how more precise measurements and predictions of life-cycle behaviour could be made.

The Programming Process

As in the Rayleigh curve theory, the new model will explain the decay in workrate on an old system in terms of the problem space becoming exhausted. For this to happen the software must be intended to provide some identifiable service. The following principle formalises that concept.

A1. A software development project has associated with it some large but finite *set of subtasks to be completed - each of* which is either a decision to be made or a problem to be solved.

Now this assumption is not intended to imply that the size of the pool can be accurately predicted in practice before development work begins. It may even be the case that the actual design chosen for the system may affect the size of the subtask pool. We avoid all these difficulties by treating the subtasks as infinitesimally small and normalising the total amount of work on each project to be one. This has the effect of focusing attention on the pattern of resource consumption rather than trying to estimate its total amount. For the latter task one would need to be able to judge the quality of acutal system designs.

To explain the increasing workrate in the initial stages of a project something more detailed than a prediction of increasing "skill" level is required. We claim that the essential mechanism has to do with how much work may be carried out in parallel. At the very beginning of the project only a small team of analysts can be used since they must identify the basic functions which will be realised by the software. The initial requirements of design team has only limited use for computing or programmer resources. It is only after the major functions and their interfaces have been identified that these can be grouped into components and passed out for further design and implementation. The key fact is that if the initial decomposition was done well, each of the components identified can be developed by separate programmers or teams of programmers working in parallel. Hence more development resources can be effectively applied and a faster rate of progress achieved as a result. Thus although management is free to follow any hiring policy, if the system is to be developed as fast as possible for a given cost, the rate of progress will be increasing during development.

In the above paragraph design of a component (specification of its input/output behaviour) was taken to always precede implementation. On a large-scale project it may not be easy to classify activities in that way. It has already been convenient to formalize the entire development task as a homogeneous space of subtasks. At this level of abstraction the natural way to indicate which problems may be tackled in parallel is to introduce a dependency relation R on the pool of subtasks.

A2. There is some partial order relation R defined on the subtasks of the project such that

$$x \; R< \; y$$

means that subtask x must be completed before work on subtask y can begin.

Relation R has to be a partial order because if there were some circular chain of subtasks

$$x1 \; R< \; x2 \; R< \; x3 \; \ldots \; R< \; x1$$

then the project could never be completed.

Progress on the project is achieved by successively solving the subtasks defined in A1. As in the opening section let W(t) denote the proportion of the project completed at time t - this being the ratio of the number of problems solved at that time to the total number of problems on the project. Then it is immediate that W should have value zero when the project starts, one when it ends and increase monotonically in between. The rate of progress on the project is dW/dt. Since the mechanism for achieving rapid progress is to work in parallel, it is reasonable to assume that more development resources are needed.

A3. The rate at which a software project consumes programming and computing resources is proportional to the rate of progress being made namely dW/dt.

Now at time t the amount of the project still unsolved and therefore needing work is 1 - W(t). But in general many of these tasks cannot be started at time t because the R-relation requires that other work be done first. Let us say that a problem is visible at time t if all its R-predecessors have been solved at that time. The following axiom formalises parallelism as the mechanism of rapid progress.

A4. The rate at which a software development can usefully consume resources is proportional to the number of subtasks which are visible and unsolved at that time.

Let V(t) denote the proportion of the total project which is both visible and unsolved at time t. Then combining axioms A3 and A4 gives

$$\frac{dW}{dt} = a. \; V(t) \qquad (3)$$

where a is some arbitrary constant.

The most interesting part of this model is concerned with characterising the rate of change of V(t). Now the only mechanism for altering V is to complete a subtask. This will certainly have the effect of removing one problem from the visible-unsolved set; it may also make some undetermined number of other unsolved subtasks visible. Clearly V depends on the rate at which problems are being solved:

$$\frac{dV}{dt} = \frac{dW}{dt} \times \left(\begin{array}{c}\text{average change}\\ \text{in V for each}\\ \text{problem solved}\end{array}\right)$$

The actual change in the number of visible unsolved subtasks when some problem is solved depends on the particular problem, the dependency relation R and which other problems have been solved at that time. If the particular subtask completed has no R-descendents then V is merely decreasedby one or, more correctly since we are working in continuous time, dV/dt=-dW/dt. But if there are unsolved R-descendants whose other R-ancestors have all been solved then these become visible and V may be increased. Clearly it is impracticalto handle each case explicitly so some averaging is needed.

But the average change for each problem solved $\Delta V(t)$ cannot be constant over the life-cycle of the project. To explain this we introduce the notion of terminal - a subtask of the project with no R-descendants. Then terminals must be more dense towards the end of the life-cycle than at the beginning. For a terminal corresponds to a problem which when solved opens up no further work. If all the early subtasks in a project were terminal then the project would quickly come to an end with no further work to be done. Conversely a project with few terminal subtasks in its later phases would never reach a proper maintenance mode. So one must expect that the density of terminals will increase as the project progresses. Consequently $\Delta V(t)$ should initially be positive (most problems have several R-descendants which become visible) and gradually turn negative as the project develops. In fact at the very end of the project one can be certain that the work being done is terminal so as $W(t) \rightarrow +1$ then $V(t) \rightarrow -1$.

A simple but powerful law with these properties is

$$\Delta(t) = c - d.W(t)^{b}$$

where b,c,d are arbitrary constants. In order to meet the limiting requirement above d=c-1. This leads to

$$\frac{dV}{dt} = \frac{dW}{dt}.\left\{c - (c+1).W(t)^{b}\right\} \quad (4)$$

which together with equation (3) defines the life-cycle pattern predicted by the new model for the development process.

Analysis of the Model

Combining equations (3) and (4) gives

$$\frac{d^2W}{dt^2} = a.\frac{dW}{dt}\left\{c - (1+c).W(t)^{b}\right\}$$

which may be integrated as it stands to give

$$\frac{dW}{dt} = a\left\{c.W - \frac{1+c}{1+b}.W(t)^{b+1}\right\}$$

This can be seen to have the solution

$$WW(t) = \frac{1}{\left[\frac{c+1}{(b+1)c} + e^{-ac(b+1)t}\right]^{1/(1+b)}} \quad (5)$$

The function defined by equation (5) has the desired property that as t becomes large and negative W(t) → 0. We also require that the total amount of work on the project is 1. This will occur if 1+c=(1+b)c which is solved by c=1/b. Hence one obtains the simpler formula for progress in development

$$W(t) = \frac{1}{\left[1 + e^{-a(b+1)t/b}\right]^{1/(b+1)}}$$

and the corresponding pattern of resource consumption or workrate is found by differentiating:

$$\frac{dW}{dt} = \frac{a}{b}.e^{-pt}.\left[1 + e^{-pt}\right]^{-\left(\frac{2+b}{1+b}\right)}$$

where p = a(b+1)/b

Comments and Conclusions

The general form of the workrate derived above is a smooth bell-shaped curve. If parameter b is chosen with a value between one and zero then the curve is skewed to the right and becomes quite similar in appearance to the Rayleigh function. It can be argued that this parameter is a measure of the extent to which a structured programming methodology was used. For large b values give dependency relations R with the terminals particularly strongly clustered toward the end. But the main tenet of improved development methodology is exactly to force the early decisions to be structural ones which define modules for subsequent implementation; details of low level implementation should be considered only later in the project; these correspond to terminal decisions. Hence the model could be used to measure whether a project succeeded in using its intended methodology.

The fact that this new model predicts life-cycle patterns approximating the Rayleigh function as a special case is an advantage since empirical evidence has been published (2) supporting the use of that function. The most obvious difference between this and the Rayleigh curve theory is that the life-cycle obtained here is defined over all time rather than just positive time. This is no great defect since for many systems it is impossible to define an exact date when the first development work relevant to that system was done.

Finally it may be possible when analysing an actual project to obtain a much more detailed picture of the dependency relation involved by constructing some form of PERT chart. This would enable a much tighter relationship milestones in development than has hitherto been possible. The accuracy of estimates of completion dates should be improved in consequence.

Bibliography

(1) P. NORDEN, 'Using Tools for Project Management', Management of Production, M.K. Starr (ed.) , Penguin Books, Baltimore, Md. 1970.

(2) L. PUTNAM, 'A Macro Estimating Methodology for Software Development', IEEE Computer Society, Compcon 1976, Washington D.C., September 1976, pp.138-143.

THE IMPLICATIONS OF LIFE CYCLE PHASE INTERRELATIONSHIPS FOR SOFTWARE COST ESTIMATING

Robert Thibodeau
E. N. Dodson

General Research Corporation
Economic Resources and Planning Operations

ABSTRACT

Past attempts to establish mathematical expressions that can predict the life cycle cost components for software systems have achieved only qualified success. The mathematical models for these relationships included only variables that describe the software characteristics and related environmental factors. This paper presents the hypothesis that software cost estimating relationships must include the effects of resources consumed in one life cycle phase on other phases. Such a model is difficult to validate. This is primarily due to the need for greater quantities of data of greater precision than is usually available. However, a preliminary result detained from existing data is positive. Therefore, additional research is justified.

INTRODUCTION

Our objective is to obtain reliable estimates of software life cycle costs suitable for initial planning. This requires the establishment of empirical relationships between life cycle cost and certain variables. We call these relationships Cost Estimating Relationships (CERs).

Our approach to establishing CERs for software life cycle costs[1,2] has been similar to other researchers. We have postulated the important relationships between the products and the amounts of resources needed to develop and operate them. We have collected project data describing resources expended and program characteristics and tried to substantiate the mathematical expressions. We have achieved relatively limited success. The CERs developed to date have relatively low precision and are applicable only to environments where resource reporting and definitions are comparable to that in which the data were obtained. Our experience is not unique. The literature shows that literally hundreds of software and environmental parameters occurring singly and in almost endless combinations have been tested in efforts to predict life cycle cost.

Our failure to obtain quantitative relationships of a precision comparable to those available for estimating the costs of hardware systems has let us to question the assumption that was implicit in our previous analyses. That assumption was that life cycle cost is predictable using variables that simply describe the product and the development environment (i.e., the type of contract, program management technique, etc.). We have formed the opinion that while all these factors are relevant, a large contribution to the resource requirements for any one phase derives from the ways in which the other phases are completed.

The way the project is executed in terms of its relationship to the planned development and other outside pressures determines *how* the resources are consumed. It also affects the quality of the delivered product.

We believe these relationships are intuitive for persons experienced in project management. We would like to develop the hypothesis of phase interrelationships by walking through two project histories that illustrate the concept. In the succeeding sections of the paper we will propose a model and show the results of applying it to actual data.

THE CASE FOR PHASE INTERRELATIONSHIPS

An important factor affecting the utilization of resources is the need to conform to a development plan. The plan is an essential management tool for ensuring that needed resources are available to the project at the proper time and in the correct amounts. We would like to see how changes in the plan, caused either by changes in requirements or by failure to meet commitments, affect cost-driving parameters. Particularly, we would like to see how management actions influence the measureable project descriptors.* Understanding this relationship should permit a more accurate analysis of the cost-driving variables.

A Project With Significant Interrelationships

Figure 1 shows the time spans and levels of effort for the different phases of a software development project. This project and the other one in this section are business data processing

*Project descriptors include man-hours for analysis, coding, testing, etc. (planned and actual), time span for the activities, numbers of personnel, application classification, etc.

systems developed by a military agency for world-wide use. Both systems were written in COBOL. Planned values are shown by solid lines and actual values by dashed lines.

Notice that there is considerable scheduled overlapping of the Design and Coding activities. This overlapping is a common practice, but it increases the likelihood that changes in the design will require parts of the system to be recoded. Such a schedule might have been adopted because time was short or because people were available only at certain times. In either case, overlapping causes any problems or delays to have increased impact on the work.

The Analysis activity of the project was carried out at about the planned level of effort.* The long delay in its completion was accompanied by some delay in the start of the Design activity. This latter delay was probably beneficial since failure to delay the start of an overlapped activity can increase scheduling problems. However, completion of Analysis lagged until three months after its scheduled date, and in the meanwhile both the Design and Coding activities were started.

The delay in the completion of Analysis indicates that some needed information was missing, some procedures were not defined, or unexpected problems occurred. Going ahead with the Design and Coding activities would almost assure an increase in the time required to code and test the system. This is a good example of one type of interaction between activities. The increased coding and testing hours would not be predicted by any method that relied solely on parameters related to those activities.

The delays, along with the substantial overlapping, were associated with a significant increase in the resources required to complete the project:

Activity	Estimated Man-Months	Actual Man-Months	Percent Increase
Analysis	4.5	6.6	47
Design	9.1	9.9	9
Coding	4.5	19.6	336
Test & Integration	4.2	10.6	152
Qualification Testing	2.2	6.0	173
	24.5	52.7	115

The figures indicate that, for this project, delays and overlapped activities were associated with large increases in the consumption of resources over what was expected. The combination of delays and parallel activities has a compounding detrimental effect on a project schedule.* For example, when coding is begun before the completion of design, the designers are required to communicate their results to the programmers in a raw, unqualified state (hence significantly increasing the chance of design errors). Overlapping also raises the possibility that the designer may not change a poor procedure when he discovers it, because he has already committed himself to the programmer. Many times the programmer may fill in missing information by himself. By doing this he may introduce errors into the system that will not be discovered until late in the testing program when repairs will be time-consuming and expensive.

We are not yet trying to build a case for cause-and-effect relationships between increased consumption of resources and delays and overlapping activities. We are simply using an existing project history to illustrate how interactions among activities may be seen to influence the expected resource requirements. On the basis of a single project one could simply conclude that the project was poorly planned and executed.

A Project With Few Interrelationships

Figure 2 shows a project that was completed in a better fashion. The Analysis and Design activities overlapped but, significantly, the design was completed before coding started. Less effort was put into the analysis than had been planned, but there was an increase in the hours required for the design:**

*We divided total man-hours by time span to determine average staffing. Therefore, any gaps in the work would reduce the calculated staffing.

*This is not to suggest that systems cannot be developed with overlapping activities. Many systems have distinct parts that can be coded before the entire design is completed. In a top-down design where coding is by tiers, the coding can often begin before the design is complete. These are planned developments that would permit the overlapping of these functions. We are concerned here with the situation where the press of the development schedule or the slippage of preceding activities results in overlapping activities that would have been accomplished better sequentially. Even in a planned implementation of parallel activities, however (and this includes top-down design), whenever the coding begins before the design is completed there is an increased risk of changes to the design or of mismatches in subsystem interfaces. The project management must weigh these risks in relation to the need for workload balancing and project scheduling.

**Many software development activities are difficult to define. The line between analysis and design becomes blurred in practice. In some instances both functions are performed by the same individual, who may also do some or all of the coding. It may be that in this instance some of the analysis hours were reported as design.

Activity	Estimated Man-Months	Actual Man-Months	Percent Increase (Decrease)
Analysis	43	24	(44)
Design	12	16	33
Coding	28	37	32
Test & Integration	12	9	(25)
Qualification Testing	3	6	100
	98	92	(6)

The project was completed on schedule.

Notice that the project described in Figure 2 was scheduled to have the Analysis phase continue until after the completion of the design activities. This occurs most often when the system specifications are not fully developed at the start of the project. Functional requirements are allowed to change during the Design phase much more than a pragmatic approach would dictate. This is the case with many information-system developments where management participation in defining functional requirements is not sufficient. As details of the design become established, the impacts of the specifications become more apparent to members of management, and their reactions require changes in the specifications. Many project managers, therefore, do not attempt to finalize the system specifications. Instead, they schedule the Analysis and Design phases concurrently. The period of analysis after the completion of the design is used to complete the documentation of the specifications.

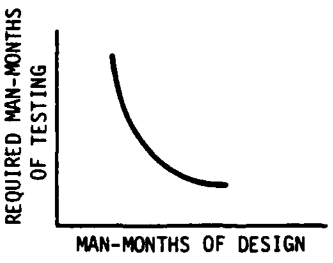
Another consequence of allowing the completion of specifications to wait until the design is completed is that the effort required to make the specification changes tends to be reported as part of the coding and subsequent activities. Analysis of project data from this point of view suggests that the practice may be quite common. Similar problems of distinguishing product and resource consumption occur in other life cycle phases. This complicates the analysis of phase interrelationships.

The preceding discussion has been presented in support of the contention that the relationships among the software development phases may be extensive and very important to the consumption of resources. As will be shown later, these relationships extend into the Operation phase.

JUSTIFICATION OF THE SELECTED MODEL

Tradeoffs Between Phases

Considering the above discussion, we are interested in proving a methematical relationship that in addition to some measure of the product resulting from the expenditure of resources contains a number of significant interrelationships among phases of the software life cycle. We want to show, for example, that a significant element in explaining the cost of testing is the effort given to the earlier activity of design.* We believe there exists--in effect-- the following type of relationship.



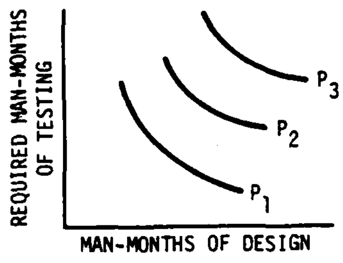
This curve indicates that--over some range--if the effort given to design is reduced, there will will be added effort required during the testing (as well as coding) phases because of errors and difficulties with the program. Conversely, additional time spent in design of the software will reduce subsequent effort required for testing (and for coding).

Similar arguments can be made about relationships between resources expended in coding or testing and the subsequent effect upon required resources after the software is installed. In these cases, man-months of coding or man-months of testing would be the abscissa and man-months of maintenance would define the ordinate.

These interrelationships pose a number of distinct analytical problems. First, the tradeoff indicated above is shown for a given program size. A simplistic attempt to correlate actual design and testing resources without including program size will almost invariably result in a positive correlation. This is because total resource requirements tend to increase with program size (and/or complexity) in a typical data base. A data base is required that is large enough to identify the variations in design and testing for a constant program size; in effect, to determine the relationships illustrated below:**

*The importance of adequate design effort is stressed throughout the literature. For example, Boehm, et.al.,[3] in an analysis of software errors found that design errors outweighed coding errors 64% to 36%. Moreover, design errors took far longer to detect and correct.

**Mathematically, these relationships can be expressed as $X_t^c = a_1 P X_d^{-b}$ (or $P = a_o X_d^b X_t^c$) where X_d = design resources, X_t = test resources, and P = program size (or complexity), i.e., a measure of output.



Where P_1, P_2, P_3 are successively larger program sizes.

	1	2	3	4	5	6	7	8
MM ANALYSIS AND DESIGN	>	=	=	<	=	>	=	=
MM CODING AND CHECKOUT	=	<	<	=	=	>	<	<
MM TESTING	=	=	>	=	=	<	>	=
MM/MO MAINTENANCE	=	=	=	=	=	=	>	>
CHANGES	NO	NO	NO	NO	YES	YES	NO	NO
REPORTED ERROR RATE								
= EQUAL TO IDEAL	<	>	=	>	>	=	>	>
> GREATER THAN IDEAL								
< LESS THAN IDEAL								

Figure 3. Postulated Trade-Offs Among Life-Cycle Man-Hour Parameters

The Proposed Model

These relationships are comparable to those discussed in basic economic tests on production theory. However, the interrelationships among the phases are actually more complex. Design, coding, testing, and maintenance are all interrelated. A limitation on design resources may be passed through all the way to maintenance. When the system is installed, the logic and coding inadequacies stemming from insufficient design become apparent as added costs for program revision. Graphically, the result is a multi-dimension tradeoff surface. Arithmetically, the relationships are of the following form:

$$P = a\ X_d^b\ X_c^c\ X_t^d\ X_m^k$$

We believe that a few relationships should dominate in this multi-dimensional set of prospective interrelationships. The most important is hypothesized to be between design and the subsequent testing period. If the results of the testing period were reasonably uniform in terms of remaining errors, then the hypothesis could be left simply between the Design and Testing phases. However, the number of errors found during the Operations phase varies widely among reported project histories. Thus, we broaden the hypothesis to state that reduced resources--relative to some norm--given to the design phase will result in greater resource requirements for testing and/or in higher error rates during the Operations phase.

Similarly, if insufficient resources are committed to the coding phase, one can expect (1) a requirement for more extensive resources during Test, and/or (2) higher error rates during the Operations phase.

Figure 3 illustrates some other departures from the ideal which may occur, and how they may be reflected in the error rate of the delivered software, which is indicative of the reliability of the software.

For example, the second column indicates that, with all other activities corresponding to the ideal and with no changes, less than ideal effort spend on coding and checkout would be expected to cause a higher error rate of the delivered software than the ideal.*

In conclusion: the development of low-risk, practical life-cycle cost estimating relationships requires the consideration of the interactions among the activities or phases. Furthermore, we postulate that any analysis that does not include these interactions will not succeed in reducing the scatter that makes existing software cost estimating schemes unsuitable for effective project planning and control.

Data Collection Problems

The interactions among life cycle phases described by the above model pose some practical problems in data collection and interpretation that must be solved before valid data will be obtained. To firmly establish the determinants of software costs, we believe that actual cost data must be recorded in keeping with a refined process model that describes the interactions that occur during system development.[1] In particular, manhours expended for, say, recoding during the nominal testing phase should be recorded as such.

*One might argue that the "ideal" error rate would be zero; but a practical solution would be to avoid spending large amounts of resources to achieve zero errors. Therefore, it would be expected that proper planning would allow for some small acceptable error rate. Obviously, this tolerance of errors does not apply to defense systems or man-rated systems, but it would be acceptable for most information systems.

An adequate cost-reporting system also requires a corresponding record of the output (e.g., lines of code) that is associated with the cost. This is one of the major problems in cost-control of software programs--it is comparatively easy to establish what costs have been incurred; the missing element is the amount of progress that has been made.

APPLICATION TO AVAILABLE DATA

There is no shortage of mathematical relationships describing software phenomena. What is in pitifully short supply is reliable data with which to prove them. Our selection of a hypothesis has increased the quantity and precision of the data required to prove it. However, we believe that no existing approach to modeling life cycle costs has been successful. Therefore, extending the complexity of the model and consequently the data requirement may be justified if they produce a better prediction of resource requirements.

It is especially difficult to obtain data describing the individual phases of the software life cyle.

Life cycle phases are not defined consistently or even accurately for most software development projects. The activities that comprise analysis as opposed to design are not clearly stated or understood by most project managers. Furthermore, most of them don't care because they have more important problems to deal with.

If a milestone concept is used in the project management, the project goes from, say, design to coding at a specified point in time. If there are changes to the design, the required effort will be reported as coding.

Even when projects are reported by activity, it is difficult to distinguish consistently among coding, unit testing, and system testing. These activities may be conducted at several levels simultaneously. Changes or detected errors can require reiterations among the activities.

As a preliminary test of the phase interrelationships model, we attempted to use some available data to describe the tradeoff between Design and Coding and Testing. Coding and Testing were combined in order to avoid the problem of distinguishing between these two phases.

Data was available from fourteen defense system projects. An equation of the form:

$$I_o = aMM_D^b \, MM_{CT}^c$$

Where I_o = Size of delivered program in 1000s of object instructions

MM_D = Manmonths of system design

MM_{CT} = Manmonths of coding and testing

a, b, c = Constants

was fit to the data. The result is shown in Figure 4.

The curves show a definite tradeoff between design effort and coding and testing. Furthermore, there is a diminishing return in reducing coding and test time obtained by increasing the design effort beyond a certain point. This point increases markedly with increasing program size. There is an indication of a very heavy penalty for failing to perform some minimum amount of design.

CONCLUSIONS

The phase interrelationships hypothesis is an intuitively satisfying model for explaining software life cycle resource requirements. Looking along this line is justified by the failure of simplier approaches.

A preliminary empirical test of the hypothesis with an extremely small data set is positive.

REFERENCES

1. Graver, C.A., et.al., Cost Reporting Elements and Activity Cost Tradeoffs for Defense System Software, CR-1-721, General Research Corporation, March 1977.

2. Dodson, E.N., et.al., Advanced Cost Estimating and Synthesis Techniques for Avionics Data Processing Software and Hardware, CR-1-701, General Research Corporation, December 1976.

3. Boehm, B.W., McClean, R.K., and Urfrig, D.B., Some Experience with Automated Aids to the Design of Large-Scale Reliable Software, IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975.

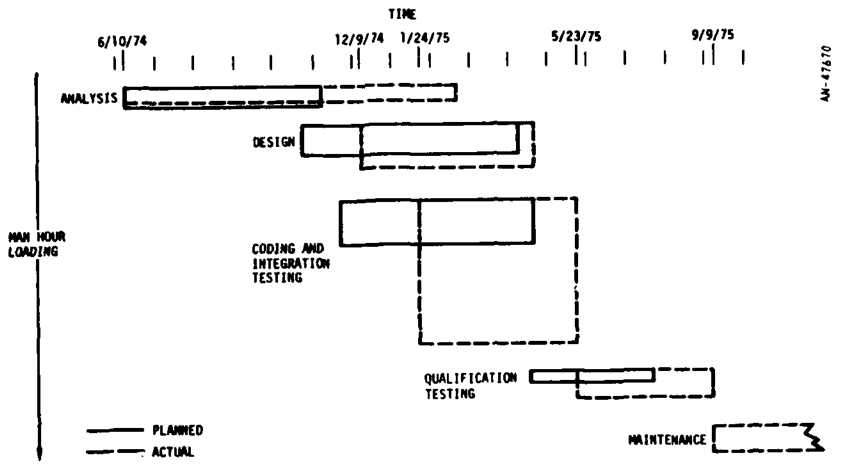


Figure 1. Scheduled and Actual Activities in a Software Development: Example 1

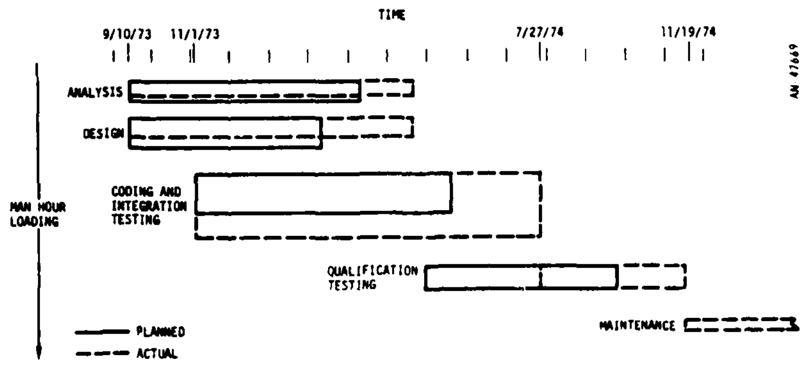


Figure 2. Scheduled and Actual Activities in a Software Development: Example 2

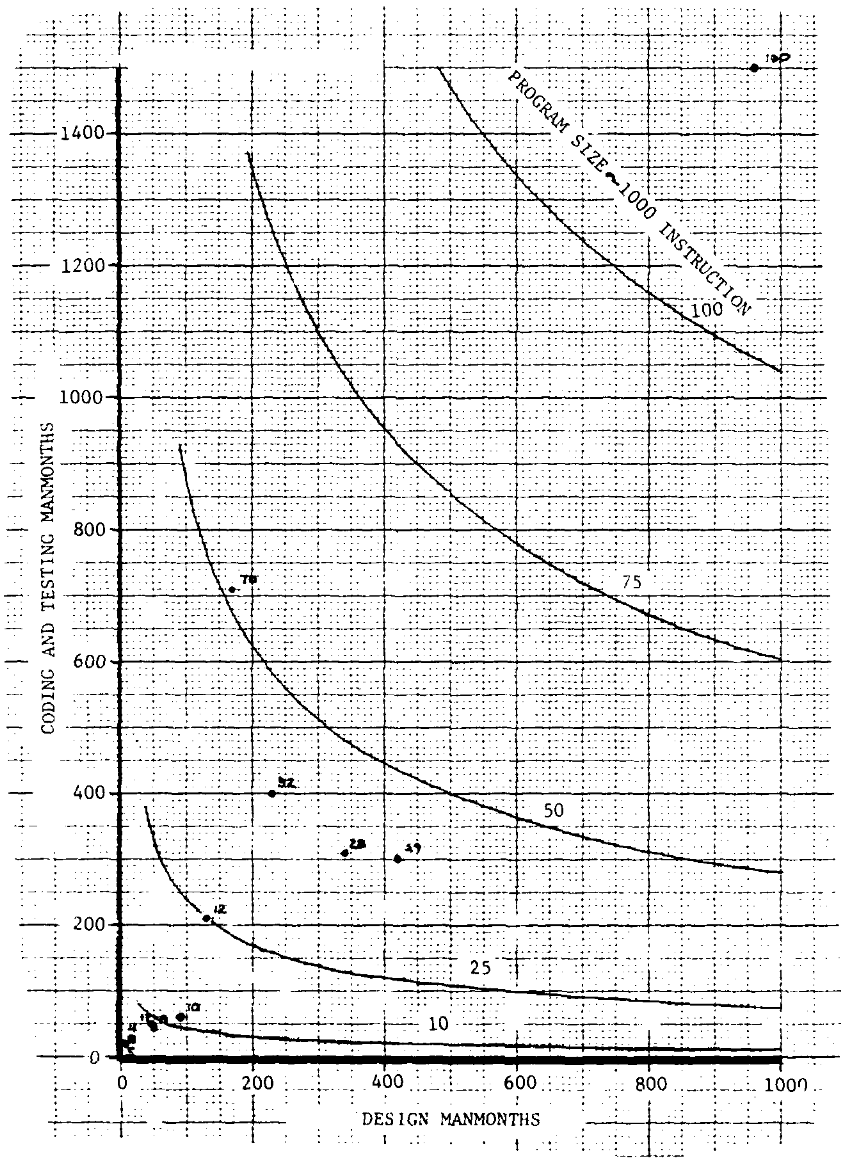


Figure 4. Relationships Between Design Effort and Coding and Testing Effort

LIFE CYCLE MANAGEMENT METHODOLOGY
DYNAMICS-PRACTICE

SUMMARY

SOFTWARE TECHNOLOGY AND SYSTEM INTEGRATION

R. C. McHenry and J. A. Rand

IBM Corporation

Gaithersburg, Maryland

ABSTRACT

Perhaps the least appreciated area of modern software technology is top-down development. Top-down development is far more than a programming technique suitable for application to an individual work assignment. Top-down development is a rich and powerful technique for project implementation and for system integration. The characteristics of the top-down process (executable as a system and self-integrating) suggest that the process may be a foundation of integration engineering.

INTRODUCTION

This paper provides key exerpts from a recent IBM technical report (FSD 78-0034) by the authors. In addition, the workshop presentation summarizes the example system employed in the full report.

A number of trends in data processing appear to justify establishing a discipline of integration engineering. These trends include:

a. Aggregating larger systems

b. Requiring higher availability

c. Distributing systems

d. Developing systems concurrently.

Mills has defined software as logical doctrine for the harmonious cooperation of people and machines (1). When a system is defined to be a coherent assemblage of people and hardware with specific capabilities, the system engineering and related operating rules are, in fact, implemented in the operational software. From this perspective, we speculate that system integration and integration engineering technology can be approached from software technology.

Our speculation can be illustrated, and perhaps substantiated, by a single approach, top-down development. Top-down development is one of the least appreciated facets of software technology. Top-down development is more than a individual work assignment technique; it is a profound technical and managerial strategy as well (2).

The key characteristics of top-down development suggest that:

a. The evolving software is always executable as a system

b. The process is self-integrating.

An adaptation of the top-down approach was conceived by O'Neill in 1972 and demonstrated in the overlapped development and integration of the TRIDENT Command and Control System (3). The concept is that software, unlike hardware, can be implemented in a system sense to be always executable and that system integration can proceed from the software (i.e., the computer) to the non-programmable hardware.

In their previous report (3), which introduced the term "integration engineering", the authors generalized the top-down adaptation by defining a top for any system and by relating the system top to an integration strategy. The top of any system is the logic for transitioning the system from state to state. This definition is general since even the simplist system must be turned on and off. In more complex systems there are many states including:

a. Initialization

b. Maintenance

c. Development

d. Training

e. Reduced function

f. Full function

g. Termination.

A key integration engineering hypothesis is that transitioning logic is a candidate integration strategy.

The need for integration engineering technology is greater than the need for systems, hardware, or software engineering technology. Far more tools exist to support design and development. In fact, implementational technology has progressed beyond our ability to effectively employ it. The continuing improvements in data processing hardware have led to the introduction of programmable hardware into many traditional non-programmable subsystems. While the resulting systems may be thought of as distributed, they often result from subsystem rather than system decisions.

OBJECTIVES

The objectives of integration engineering will be realized by an approach that places state transitioning at the top of the system. Component boundaries for integration are not drawn between hardware and software. The boundaries are drawn around analyses that define the system states and the state transitioning. Maintenance, training, and reduced function states thereby become an integral part of the design and development process, and appear as early milestones on the integration schedule.

Early deliveries are aimed at integrating the lowest operating level (i.e., the maintenance state): the configuration with the minimum available hardware. Each successive delivery (and integration) can be viewed as a transition from one operating state to the next higher one. After integration of the final deliveries (full up state), the preceding deliveries are retained rather than discarded.

Delivery of a maintenance state allows for integration of hardware/software from a component level through the subsystem to the system level with continuous availability of system test elements. This allows for verification of performance at each level of integration without dependence on higher levels performing correctly.

APPROACH

Some underlying concepts of integration engineering are presented in the following paragraphs.

Top-Down Implementation

The top-down approach is patterned after the natural approach to system design and requires that programming proceed from developing the control architecture (interface) statements and data definitions downward to developing and integrating the function units. Top-down programming is an ordering of system development which allows for continual integration of the parts as they are developed and provides for interfaces prior to the parts being developed.

Operating Modes

Operating states or modes describe the various configurations of system resources (people, programs and equipment) which operate the system. Preoccupation with the full operational function state can lead to an unsatisfactory system since the early work must also specify the procedures required to transition the system from state to state. The requirements determination process which precedes system development must specify the functional availability (e.g., tolerable outage) requirements to guide the eventual development of procedure configurations for each state.

Each state has instrinsic procedures and exists in a broader procedural environment which enables transition from state to state. A system top is the transitioning procedure and this perspective helps drive the entire process (function, development, test, operation).

Prototypes

Prototype systems are developed for concept validation, feasibility determination, or benchmark calibration purposes. The sussessful prototype generally represents an early step in a major system acquisition. While the successful prototype overcomes the certainty of specific failures, the typical prototype does not guarantee full scale success. Serious problems or disappointing performance may arise in scaling up to an operational system. Consideration must be given to scaling problems throughout the prototype planning.

One approach to prototyping is to scale down from the system design to the prototype design. In an evolutionary development approach, the prototype is a subset of the intended system. The prototype may, in fact, be an operational state of the intended system.

Integration

Since software, unlike hardware, can be implemented in a system sense to be always executable, the system integration should proceed from software (i.e., the computer) to the nonprogrammable hardware. Phased deliveries of software to software/hardware integration ensure intermediate evaluations where managers can determine progress and acceptability of concurrently developed components. Phased deliveries provide the means of integrating hardware and software incrementally rather than as one enormous task late in the project. The problems are found and corrected earlier while more schedule is available for retesting.

The test system should also be built incrementally by phasing its development to match the system test and integration phasing.

Earlier, operating states were described and the state transitioning procedures were considered to be a top of any system. A process is postulated to exploit the transitioning procedures as a strategy for implementing and integrating the larger than software system. The process considerations are:

AD-A104 212 INTERNATIONAL BUSINESS SERVICES INC WASHINGTON DC F/G 9/2
SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP (2ND) AUGUST 21-22, 197--ETC(U)
AUG 78 V R BASILI, E H ELY DAAK70-78-D-0030
UNCLASSIFIED 78CH1390-4C NL

2 OF 3
ADA
104212

a. The number of states and transitions required for integrating the system may exceed the number required for operating the system.

b. The testing of reduced function states is progressive and is an integration process by-product rather than an afterthought to full function state testing. The more critical functions (i.e., those functions occurring in more states) receive more testing.

c. Where development funding is spread over many years (or under design-to-cost limits) the evolving system can be executed as a reduced system for testing or limited operation.

Incremental Development

The top-down approach is applied in discrete stages (initial, intermediate, final) to improve manageability. Each stage is carried to readiness for system integration.

The initial stage begins early in the schedule, after the design specifications are developed, to exercise and validate the interface between executive and functional software and to obtain an early validation of computer utilization allocations.

The intermediate stages are developed to exercise and vaildate the functional software interfaces and selected critical system functions. Simulation software provides inputs to the functional software in lieu of actual hardware inputs. Intermediate stage software is used for interface validation and early system hardware integration.

The scheduling of initial and intermediate stages must provide time for appraisal and redirection of development. The final stage is developed to provide all functions the software is required to perform.

Software integration and verification is a controlled process by which intermediate and final deliveries are integrated in simulation environments which at successive tests more fully approximate actual use. The objectives are to verify that the integrated software will perform its specified functions in the simulated environment and to reduce the number of problems encountered during system test and evaluation.

System Readiness Functions

In complex systems, the various assessment and hardware maintenance support functions must be integrated into the operational system. These functions include:

a. Interface tests to assess hardware integration during operation and testing

b. Operability tests to assess functional availability

c. Equipment tests to assess availability and to support corrective or preventive maintenance

d. Component status determination to support reconfiguration

e. Data extraction for presentation or subsequent analysis.

In addition, the personnel readiness functions (training and exercise) can be integrated into the operational system.

Applicability to Subsystems

The analytical process by which a system is partitioned into states can be applied to subsystems and lower level subdivisions. A result of the process is the transitioning procedure which employs system parameters in its conditional logic. While we may consider any conditional expression as choosing among states, at some point we cease to distinguish states unless a software/hardware/people reconfiguration or major component state change is involved. At the system level we may wish to consider intra-subsystem states as sub-states. (Sub-states have the property of being nested within states).

Now if we consider testing and more particularly integration to be conditional in nature, we may consider test and integration as transitioning procedures. Similarly, incremental development may be considered in a transitional context.

TRANSITION MANAGEMENT

A key hypothesis of integration engineering is that the state and transition management is a natural part (i.e., the top) of the system and should not require schedule extensions for testing. Transition management is the implementation of the integration engineering apprcach stated under APPROACH. Some work remains to be done in order to generalize transition implementation. The issues include the degree to which:

a. Transition management is dependent on the operating system

b. The transition management function is distributed in a distributed system

c. The actual configuration/state is transparent to the application functions

d. System readiness functions are implemented.

Transition management, like integration engineering, exposes most system design issues from the basic system architecture to component reliability requirements. However, every system is in some way unique in the parameters that define transition management.

Operating System

Transition management can be viewed as part of, a superset of, or a subset of the operating system. In practice transition management is all three. If, however, the operating system interface to the functions required by transition management (e.g., system loader, fault recovery, interrupt processing) is well defined and documented, then the degree of dependence would be minimized.

Distributed Management

The transition management function is completely dependent on the system architecture. For example, in a distributed system the transition management function itself could be distributed. The issues raised here are the same as those raised in connection with a distributed data management system (e.g., master/slave versus peer to peer, dead-lock avoidance, functional integrity). The resolution of these issues leads directly to questions about global status data, information hiding, and the connectivity between nodes.

Application Functions

Obviously, the application functions must be designed to be independent of the system configuration. This is, however, achievable in any system and made easier by functional bus addressing (4).

System Readiness

The system readiness functions are the single most important element to transition management, and one of the critical tools for integration engineering. If the system has the flexibility and redundancy to totally (or partially) recover from a single failure, but cannot detect the failure when it occurs, then the system flexibility is wasted.

The system readiness functions must provide fully integrated, online system tests and equipment diagnostics that operate in realtime. The function should continually assess system readiness to meet the defined mission while causing minimum degradation to the operational system. The readiness functions should include the following tests:

a. Operability tests assess the capability of the entire system to perform as required. The test should address the system as a whole and verify that the various complete functions can be performed.

b. Interface tests assess the capability of all hardware units to intercommunicate properly and to perform their electrical functions correctly. This test should address each individual hardware unit and all of its signal interfaces. Detected hardware unit failures should be immediately displayed.

c. Alignment tests assist in the assessment of sensor and/or system alignment parameters.

d. Performance monitoring tests check each equipment unit and interface for proper operation.

e. Fault location tests exercise the unit to make a detailed determination of the exact nature and location of the failure for repair. They require that the subject equipment be dedicated to testing and repair.

These tests are required to operate "online" in that they will run concurrently with the operational program without any system degradation, as opposed to an "offline" test which is conducted independently of the operational program and prohibits that program from meeting its designated operational requirements.

REFERENCES

1. H.D. Mills, "Software Engineering," Science, Vol. 195, No. 4283, March 1977.

2. C. L. McGowan and R. C. McHenry, "Software Management," Research Directions in Software Technology, MIT Press, 1978.

3. R. C. McHenry and J. A. Rand, "Integration Engineering: An Approach to Rapid System Deployment," FSD 77-0179, IBM Corporation, Gaithersburg, 1977.

4. R. C. McHenry and J. A. Rand, "Software Technology and Integration Engineering," FSD 78-0034, IBM Corporation, Gaithersburg, 1977.

ESTABLISHING A SUBJECTIVE PRIOR DISTRIBUTION FOR THE APPLICATION OF LIFE CYCLE MANAGEMENT FOR COMPUTER SOFTWARE

G.J. SCHICK
and
CHI-YUAN LIN

UNIVERSITY OF SOUTHERN CALIFORNIA

ABSTRACT

In the development of large scale computer software and in the management of the development process it is often useful to model the reliability and/or the cost of development of these software packages. The literature has many references that assume a model and show its usefulness as a management tool. The reader is referred to references [29 through 41]. Several of these publications use Bayesian methodology. There are divided opinions as to the application of some of these models, e.g. some authors feel the exponential distribution found so useful in the reliability field for hardware should not be applied to software. Others vehemently disagree. In fact some even show data that seem to substantiate their belief. The papers relating to Bayesian methodology also assume a prior distribution [34,35,36,37]. These assumptions in turn can also be challenged. In order to get an idea of what type of a probability distribution might be applicable and what mathematical form might be appropriate we offer here a structured approach in assessing the probability distribution subjectively. It is possible to base the analysis on either (1)the subjective prior distribution when no test data are available, or on (2) a posterior distribution which with the use of Bayes' Theorem, combines the prior distribution with the likelihood function (the sampling evidence).

It is probably true that the engineering practitioners by and large are not familar with Bayesian statistical concepts. Of course, there are exceptions. This paper offers a methodology of assessing a prior distribution subjectively. Once this has been done the general shape of the distribution can be ascertained,then the search for the mathematical form is greatly simplified. For instance, the probability distribution may be skewed, not exist for negative values of the random variable. This would eliminate a whole series of probability models like e.g., the normal distribution and give rise to a host of others. While it is still possible to select a model from many available basic models, the selection process is at least based upon some evidence, namely the opinion of the experts in charge of developing the software package. Two computer programs were written.

a. Assessing a subjective prior distribution by elicitating answers to questions on a CRT (Cathode Ray Tube). The answers to these questions are used to plot the distribution function as well as the density function.

b. From the general shape of these functions a family of probability functions are suggested. For instance an inverted gamma distribution, a beta distribution, or say a log normal distribution might be hypothesized. Some of the summary output of the first program become inputs for finding the parameters of the assumed distributions

An example,of a log normal distribution is used but other families of distributions could have been selected as well.

This paper does not explicitly deal with the derivation of the posterior distribution which is found via Bayes' Theorem in conjunction with incoming data. The prior distribution, however, is an essential part of finding the posterior distribution. If the prior distribution found is integrated with test information as data become available, then obviously this is more complete information than just test information alone.

INTRODUCTION

The importance of consistent prior distributions is two-fold. First, these distributions reflect consistent initial predictions because they are developed by a structured process. Second, these distributions are the starting point for applying Bayes' Theorem to develop the posterior distribution by modifying the prior distribution with actual data available later.

This paper will show how a prior distribution can be found subjectively, even though no collateral data are available. Then, once this has been achieved, a family of known probability functions is used to ascertain if the found prior distribution belongs to the given family. This paper does not address the subject of determining the posterior distribution.

Two interactive computer programs were written.

a. Subjective assessment of fractiles.

b. Using some of the fractiles found in (a), the parameters of the log normal distribution are found. A log normal distribution has been selected as an example but any other pertinent distribution could have been used.

PREVIOUS WORK

More recently, decision theory has been considered as a general framework for logical analysis of a decision problem under uncertainty. As such, considerable attention has been given to problem formulation and methods for the assessment of a prior distribution. For example, Schlaifer's recent book [22] is largely devoted to the formulation and prior analysis of decision problems; posterior analysis is discussed only in the last part of the text. Howard and his associates (see, for example, [11] and [25]) have emphasized the application of decision theory to complex, dynamic, and uncertain decision problems. In dealing with these problems, they have explicitly included the problem formulation phase in the decision analysis cycle.

Decision theory, either concerned with specific models or general frameworks, treats uncertainty through subjective probability and treats attitude toward risk through utility theory. Regardless of whether the decision maker is concerned with prior or posterior analysis, the prior probability distribution, reflecting his quantified judgments about uncertainty, is an indispensable input to the analysis.

One difficulty associated with probability assessment is the assessor's inconsistencies which often occur in formulating a prior distribution. The question of how to discover and remove inconsistencies is of general interest to decision analysts. Another question of interest is how to fit a probability distribution using the assessed fractile in order to make the subsequent analysis more tractable. Both of these questions are addressed in this paper. The paper offers two computer programs. The first program allows a person to interact with the computer via a graphical device (Cathode Ray Tube [CRT]) during his course of establishing a subjective distribution. The second program fits a lognormal distribution to the subjective distribution.

During recent years, subjective probability has been studied by researchers in various disciplines such as psychology, mathematics, statistics, engineering, and business administration (as evidenced by the references at the end of the paper). While some of these studies are mainly theoretical or philosophical, others are experimental.

In their text [17], Pratt, Raiffa, and Schlaifer present the method of equally likely subintervals. Subsequently, Raiffa [18] illustrates this method in detail by providing a dialogue between a decision analyst and his client. Schlaifer [22] advocates this method and offers a computer program for fitting a cumulative function through assessed fractiles.

For his experimental study, Winkler [26] developed a questionnaire using four assessment techniques: (a) Cumulative Distribution Function – assessment of fractiles by means of equally likely subintervals or direct questions regarding fractiles, (b) Hypothetical Future Samples, (c) Equivalent Prior Sample Information, and (d) Probability Density Function. He used this questionnaire to elicit prior distributions from 38 selected subjects involved in his study.

The use of penalty functions, or scoring methods, has been discussed by several researchers as means of encouraging honest assessments. Specifically, de Finetti [3] presents the quadratic scoring rule. Savage [21] derives the general class of strictly proper scoring rules by considering probabilities as special cases of rates of substitutions. Winkler discusses the use of scoring rules and other payoff schemes [27] and reports his experimental results [28].

Staël von Holstein and his associates ([24] and [25]) focus on the subject of eliciting the opinions of experts in practical situations rather than laboratory experiments. They discuss probability encoding in the context of decision analysis and propose the use of a probability wheel to facilitate the encoding process.

At the Reliability Conference in 1970, Lin and Schick [13] presented the use of an on-line computer system to assist a person in developing a prior distribution to represent his beliefs. Although the console-aided procedure is illustrated by a problem in the reliability field, this procedure is applicable to assessment of any prior distribution. Since then, considerable experience with this procedure has been gained from experiments involving students in several statistics and decision theory classes at the University of Southern California.

The present paper results from the authors' continued effort in making the probability assessment more practical by using modern electronic computers. This paper offers a newly designed computer program which has incorporated the experience gained from the use of the previous program. To simplify the assessment procedure, the new program: (a) reduces the number of questions significantly (from 12 to 6), (b) is highly conversational and interactive, (c) checks for consistency as the user answers question by question, (d) uses graphical display rather than the typewriter terminal to help the user visualize the assessment process as well as to greatly increase the speed of drawing the assessed probability curves, and (e) plots not only the cumulative function but also the density function. Once a subjective distribution has been determined, a second computer program will fit a lognormal distribution to the subjective distribution to make the subsequent analysis of debugging problems more tractable mathematically.

METHOD OF ASSESSMENT

Several methods have been suggested for estimating prior distributions (see, for example, [9], [17], and [26]). Our computer program makes use of the method of equally likely subintervals, which perhaps is the most commonly used approach. The basic idea of this method is to ask the decision maker, at any stage, to divide a given interval into two judgmentally equally likely subintervals.

To begin with, the interval covering all possible values of an uncertain quantity (usually called a random variable) is split into two subintervals and the decision maker is asked to choose which subinterval to bet on. The dividing point is then changed until he feels indifferent between betting on one or the other subinterval. When the indifference point is reached, the decision maker feels that it is equally likely that the actual value of the uncertain quantity will fall above (to the right of) or below (to the left of) this point. The indifference point, which divides the entire interval into two subintervals with equal probabilities, is the median. Next, the decision maker is asked to specify a point which will further divide the subinterval to the left of the median into two equally likely parts. This new point is the first quartile. Similarly, the subinterval to the right of the median may be further divided into two equally likely parts. The decision maker may proceed in this manner to divide any given interval (generated previously) into two equally likely subintervals.

Suppose we let x_k designate the k^th fractile of the uncertain quantity $\tilde{x}$, i.e.,

$$P(\tilde{x} \leq x_k) = k \ , \qquad 0 < k < 1$$

Then, using the method of equally likely subintervals, the decision maker is asked to respond to a series of questions which will lead to a determination of x_k values for such k as 0.5, 0.25, 0.75, etc.

COMPUTER PROGRAM

The program stores a set of questions for the method of equally like subintervals. These questions are displayed successively on a CRT, the user responds to the questions by typing his answers on a teletype. The response to each of the questions is processed immediately and checked for logical consistency.

Assuming you are the user of the program, the first question calls for the lower limit of the probability distribution by asking you to:

> "Specify the largest value such that you feel virtually certain that the actual value of the uncertain quantity will fall above this value."

The second question, on the other hand, calls for the upper limit of the distribution by asking you to

> "Specify the smallest value such that you feel virtually certain that the actual value of the uncertain quantity will fall below this value."

In terms of the fractile notation described earlier, the first question asks for x_0 and the second question asks for x_1. The program will check to see if x_0 is less than x_1 and if you feel virtually certain that the actual value of the uncertain quantity will lie in between x_0 and x_1.

The third question asks you to divide the interval defined by the limits x_0 and x_1 into two equally likely subintervals. The question says:

> "Specify the value such that you feel it is equally likely that the actual value of the uncertain quantity will fall above or below this value."

The answer to this question yields $x_{0.5}$, which should lie in between x_0 and x_1.

The fourth question, which calls for $x_{0.25}$, is:

> "Suppose you were told that actual value is less than $x_{0.5}$. Specify the value such that it is equally likely that the actual value of the uncertain quantity is either above or below this value."

The program will check to see if this answer lies in between x_0 and $x_{0.5}$.

The fifth question, which calls for $x_{0.75}$, is:

> "Suppose you were told the actual value is greater than $x_{0.5}$. Specify the value such that it is equally likely that the actual value of the uncertain quantity is either above or below this value."

This answer is checked to see if it lies in between $x_{0.5}$ and x_1.

At this point, the program further checks for consistency. Specifically, it asks:

> "Now, do you feel it is equally likely that the actual value of the uncertain quantity will lie within the interval between $x_{0.25}$ and $x_{0.75}$ or outside of this interval?"

If the check is not met, the program will direct you to review and revise each of your previous answers. Otherwise, the program will proceed to ask you to specify the most likely value (the mode).

The assessments thus obtained are summarized on the CRT. The program then fits a smooth cumulative distribution function through the assessed fractiles. At your request, it will plot the cumulative curve and the corresponding density curve. If these graphs do not seem to reflect your judgments about the uncertain quantity, you will be guided by the program to revise your previous responses. Whenever you are satisfied with the assessed distribution, the mean and the standard deviation are computed. In addition, you may ask for 0.005, 0.015, 0.025, . . . , 0.995 fractiles of the distribution.

COMPUTER OUTPUT

To illustrate the computerized method of probability assessment discussed above, the computer output of an example is presented. In this example, the expert is asked to quantify his judgments concerning the debugging hours for a particular job. As we can see from this output, the expert violates some of the probability axioms and is asked to revise his responses several times.

```
THIS PROGRAM IS DESIGNED TO ASSIST YOU IN (A) QUANTIFYING YOUR
PROBABILITY JUDGMENTS CONCERNING AN UNCERTAIN QUANTITY,
(B) CALCULATING THE MEAN AND STANDARD DEVIATION OF THE PROBABILITY
DISTRIBUTION OBTAINED FROM THIS QUANTIFICATION, AND (C) FITTING THE
ASSESSED DISTRIBUTION TO A THEORETICAL DISTRIBUTION

WHAT IS THE UNCERTAIN QUANTITY OF YOUR CONCERN NOW?

    NUMBER OF DE BUGGING HOURS

PLEASE RESPOND TO THE FOLLOWING QUESTIONS WITH YOUR CAREFUL
JUDGMENTS

   (1) SPECIFY THE LARGEST VALUE SUCH THAT YOU FEEL VIRTUALLY
       CERTAIN THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY
       WILL FALL ABOVE THIS VALUE

       650

   (2) SPECIFY THE SMALLEST VALUE SUCH THAT YOU FEEL VIRTUALLY
       CERTAIN THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY
       WILL FALL BELOW THIS VALUE.

       100

SORRY, YOU HAVE MISINTERPRETED THESE TWO QUESTIONS. QUESTION (1)
ASKS FOR THE LOWER LIMIT OF THE UNCERTAIN QUANTITY. PLEASE GIVE YOUR
NEW ANSWER TO (1)

       100

QUESTION (2) ASKS FOR THE UPPER LIMIT OF THE UNCERTAIN QUANTITY.
PLEASE GIVE YOUR NEW ANSWER TO (2).

       650

YOUR ANSWERS TO THESE TWO QUESTIONS IMPLY THAT YOU FEEL VIRTUALLY
CERTAIN THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY WILL LIE IN
BETWEEN 100 AND 650. DO YOU AGREE?

       NO

THEN YOU MUST REVISE YOUR ANSWER(S)

PLEASE TYPE YOUR NEW ANSWER TO (1).

       50

PLEASE TYPE YOUR NEW ANSWER TO (2)

       650

   (3) SPECIFY THE VALUE SUCH THAT YOU FEEL IT IS EQUALLY LIKELY
       THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY WILL FALL
       ABOVE OR BELOW THIS VALUE

       250

   (4) SUPPOSE YOU WERE TOLD THAT THE ACTUAL VALUE IS LESS THAN 250.
       SPECIFY THE VALUE SUCH THAT IT IS EQUALLY LIKELY THAT THE
       ACTUAL VALUE OF THE UNCERTAIN QUANTITY IS EITHER ABOVE OR
       BELOW THIS VALUE

       310

*USER'S RESPONSE APPEARS IN ITALICS
```

I BELIEVE YOU DIDN'T ANSWER THIS QUESTION CAREFULLY. PLEASE GIVE IT SOME THOUGHT.

DO YOU NOW REALIZE WHAT YOU DID WRONG?

NO

YOUR ANSWER TO (4) SHOULD HAVE BEEN IN BETWEEN 50 AND 250. PLEASE TYPE YOUR NEW ANSWER.

130

(5) SUPPOSE YOU WERE TOLD THAT THE ACTUAL VALUE IS GREATER THAN 250. SPECIFY THE VALUE SUCH THAT IT IS EQUALLY LIKELY THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY IS EITHER ABOVE OR BELOW THIS VALUE.

400

(6) NOW, DO YOU FEEL IT IS EQUALLY LIKELY THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY WILL LIE WITHIN THE INTERVAL BETWEEN 130 AND 400 OR OUTSIDE OF THIS INTERVAL?

NO

SORRY, YOUR NO ANSWER INDICATES THAT YOUR ANSWERS TO QUESTIONS (3) THROUGH (5) MAY NOT REFLECT YOUR PROBABILITY JUDGMENTS. SO, LET'S REVIEW EACH OF THESE ANSWERS.

(3A) ACCORDING TO YOUR ANSWER TO QUESTION (3), YOU FEEL IT IS EQUALLY LIKELY THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY WILL FALL ABOVE OR BELOW 250. DO YOU AGREE?

YES

(4A) WE NOW LOOK AT YOUR ANSWER TO QUESTION (4). IF THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY WERE LESS THAN 250, WOULD YOU FEEL EQUALLY LIKELY THAT THE ACTUAL VALUE IS EITHER ABOVE OR BELOW 130?

NO

PLEASE TYPE YOUR NEW VALUE FOR WHICH YOU WOULD FEEL EQUALLY

180

(5A) FINALLY, YOUR ANSWER TO QUESTION (5) IMPLIED THAT IF THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY WERE GREATER THAN 250 YOU WOULD FEEL EQUALLY LIKELY THAT THE ACTUAL VALUE IS EITHER ABOVE OR BELOW 400. DO YOU STILL AGREE?

YES

(6A) NOW, LET'S CHECK THE CONSISTENCY OF YOUR REVISED ANSWERS. THE LOGICAL CONSISTENCY REQUIRES YOU TO FEEL EQUALLY LIKELY THAT THE ACTUAL VALUE OF THE UNCERTAIN QUANTITY WILL LIE WITHIN THE INTERVAL BETWEEN 180 AND 400 OR OUTSIDE OF THIS INTERVAL. DO YOU FEEL THAT WAY?

YES

(7) SPECIFY THE MOST LIKELY VALUE (THE MODE)

200

(8) GREAT. YOU NOW HAVE DONE YOUR ASSESSMENTS AS SUMMARIZED

CUM. PROB.	VALUE	CORRESP QUESTION
0.00	50	1
0.25	180	4
0.50	250	3
0.75	400	5
1.00	650	2
MODE	200	7

DO YOU WISH TO REVISE YOUR ANSWERS?

NO

FROM THESE ASSESSED POINTS, A SMOOTHED CUMULATIVE DISTRIBUTION WILL BE OBTAINED. WOULD YOU LIKE TO SEE THE GRAPHS FOR THE DISTRIBUTION FUNCTION AND THE DENSITY FUNCTION?

YES

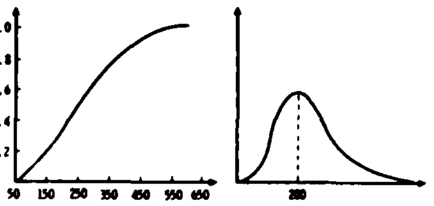


Figure 1. Distribution and Density Functions for De-Bugging Hours

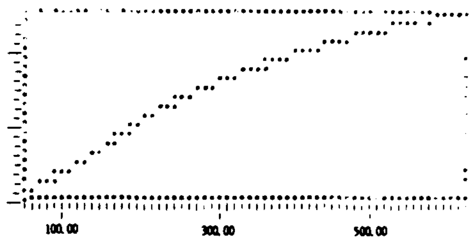


Figure 2. Cumulative Distribution

(9) WOULD YOU ACCEPT THESE GRAPHS AS REASONABLY ACCURATE EXPRESSIONS OF YOUR JUDGMENTS CONCERNING THE UNCERTAIN QUANTITY?

YES

(10) SUMMARY MEASURES OF THIS DISTRIBUTION ARE

MEAN	284.9094
STANDARD DEVIATION	155.4427

(11) DO YOU WANT TO SEE THE VARIOUS FRACTILES?

YES

```
0.005  54.[illegible]  **  0.255  161.780  **  0.505  252.420  **  0.755  403.[illegible]
0.015  60.[illegible]  **  0.265  165.204  **  0.515  257.[illegible]  **  0.765  410.872
0.025  64.400  **  0.275  168.900  **  0.525  262.[illegible]  **  0.775  418.200
0.035  68.104  **  0.285  172.404  **  0.535  267.780  **  0.785  425.[illegible]
0.045  73.776  **  0.295  175.876  **  0.545  273.220  **  0.795  433.[illegible]
0.055  78.416  **  0.305  179.316  **  0.555  278.820  **  0.805  440.[illegible]
0.065  83.024  **  0.315  182.724  **  0.565  284.576  **  0.815  448.152
0.075  87.300  **  0.325  186.100  **  0.575  290.400  **  0.825  456.000
0.085  92.144  **  0.335  189.444  **  0.585  296.296  **  0.835  463.714
0.095  96.[illegible]  **  0.345  192.756  **  0.595  302.144  **  0.845  472.[illegible]
1.105  101.128  **  0.355  196.036  **  0.605  308.[illegible]  **  0.855  480.[illegible]
0.115  105.[illegible]  **  0.365  199.284  **  0.615  314.016  **  0.865  490.234
0.125  110.000  **  0.375  200.000  **  0.625  320.000  **  0.875  500.000
0.135  114.[illegible]  **  0.385  203.616  **  0.635  326.016  **  0.885  510.[illegible]
0.145  118.872  **  0.395  207.204  **  0.645  332.084  **  0.895  520.320
0.155  122.912  **  0.405  210.944  **  0.655  338.144  **  0.905  530.720
0.165  127.[illegible]  **  0.415  214.[illegible]  **  0.665  344.288  **  0.915  541.200
0.175  131.200  **  0.425  218.400  **  0.675  350.400  **  0.925  552.[illegible]
0.185  135.240  **  0.435  222.170  **  0.685  356.570  **  0.935  562.[illegible]
0.195  139.232  **  0.445  226.020  **  0.695  362.820  **  0.945  573.920
0.205  143.162  **  0.455  230.020  **  0.705  369.220  **  0.955  585.120
0.215  147.[illegible]  **  0.465  234.100  **  0.715  375.760  **  0.965  596.400
0.225  150.000  **  0.475  238.[illegible]  **  0.725  382.500  **  0.975  608.000
0.235  154.520  **  0.485  242.[illegible]  **  0.735  389.[illegible]  **  0.985  619.[illegible]
0.245  158.192  **  0.495  247.820  **  0.745  396.420  **  0.995  631.[illegible]
```

(12) DO YOU WANT TO FIT THE ASSESSED DISTRIBUTION TO A THEORETICAL DISTRIBUTION?

NO

(13) DO YOU WISH TO QUANTIFY YOUR JUDGMENTS CONCERNING ANY OTHER UNCERTAIN QUANTITY?

NO

THANK YOU FOR YOUR COOPERATION. GOOD-BYE

AN APPLICATION FROM PROGRAM VERIFICATION

From the assessment procedure given earlier, several fractile points, the mean, and the standard deviation are available in the summary output of the computer program. Any two fractile points, or a fractile point and the mean, or a fractile point and the standard deviation etc., can be used to determine the parameters of the lognormal distribution. A new program was developed that allows some 20 different input combination pairs in the procedure for determining the parameters of the lognormal distribution. The density function of the lognormal distribution is given by:

$$f(x) = \frac{1}{\beta\sqrt{2\pi}}\, x^{-1} \exp\left[-\frac{1}{2}\left(\frac{\ln x - \alpha}{\beta}\right)^2 \right], \quad \begin{matrix} x > 0 \\ \beta > 0 \end{matrix} \qquad (1)$$

where α and β are the parameters of the lognormal distribution.

It is well known that the mean E(x) and the variance V(x) are given by:

$$E(x) = \mu = \exp(\alpha + [1/2]\,\beta^2)$$

$$V(x) = \sigma^2 = \mu^2 \left[(\exp \beta^2) - 1 \right]$$

The mode of this distribution is at

$$MODE = \exp\left(\alpha - \beta^2\right)$$

while the median or 50th percentile, P_{50}, is at

$$P_{50} = e^{\alpha}.$$

By letting $y = \frac{\ln x - \alpha}{\beta}$ in (1) and using standard normal tables, the 90th percentile was found to be

$$P_{90} = \exp(1.282\beta + \alpha).$$

Other fractile points can be found in a similar fashion.

As we have seen the lognormal distribution has two parameters α and β. Thus to fit a lognormal distribution to the subjectively derived distribution we only have to specify two values such as P_{50} and P_{90}, or the mean and the standard deviation. For the following example the mode = 200 and the median = 250 are used. The program output includes a distribution function and a density function. The latter is given in Figure 3.

```
                    LOG NORMAL DISTRIBUTION


      DO YOU NEED THE COMBINATION PRINTOUT? YES=1, NO=0 ?0


          WHAT IS THE INPUT COMBINATION NUMBER ?13
          MEDIAN = ?250
          MODE = ?200


                                                         90TH
  ALPHA   BETA     MEDIAN     MEAN      STD DEV   MODE       PCTLE TIME
  5.5215  0.4724   250.0000   279.5086  139.7542  200.0000   458.0842


* DO YOU WISH TO INTEGRATE-NO=0, YES=1, RETURN=2 ?0
```

```
DO YOU WISH TO PRINT X AND Y-NO=0, YES=1, RETURN=2 ?1
WHAT IS XMIN,XMAX,DELX
?100,650,20


X-VALUES          Y-VALUES           X-VALUES          Y-VALUES
100               1.28704E-03        400               1.28704E-03
120               2.10476E-03        420               1.10016E-03
140               2.84011E-03        440               9.37937E-04
160               3.37814E-03        460               7.98064E-04
180               3.68409E-03        480               6.78084E-04
200               3.77688E-03        500               5.75580E-04
220               3.7078E-03         520               4.88276E-04
240               3.50578E-03        540               4.14090E-04
260               3.23704E-03        560               3.51159E-04
280               2.93064E-03        580               2.97842E-04
300               2.61306E-03        600               2.52707E-04
320               2.30232E-03        620               2.14516E-04
360               1.74150E-03        640               1.82209E-04
380               1.50047E-03

DO YOU WISH TO PLOT X AND Y; NO=0, YES=1 ?1
```

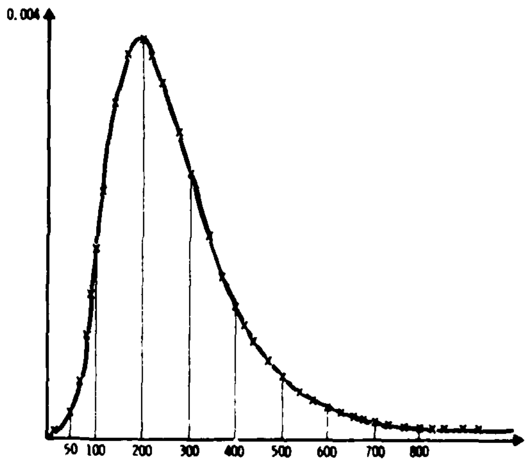


Figure 3. Lognormal Density Function with Median = 250 and Mode = 200

Now the distribution function or density function can be visually compared with the subjectively derived prior distribution using the questionnaire involving the debugging hours. If "reasonable" agreement has been achieved the mathematical form of the density has been found. Several combinations of input values might have to be examined in order to achieve the "best" fit. This form is important in order to establish the posterior distribution using incoming data and the likelihood function according to Bayes' Theorem. On the other hand, if "reasonable" agreement between the two distribution functions has not been achieved, a new family of distributions may be tried and/or the empirical distribution might be questioned. Ultimately, agreement will be found unless the lognormal distribution is not a valid model

REFERENCES

1. M. Alpert and H. Raiffa, A Progress Report on the Training of Probability Assessors, *Unpublished manuscript*, Harvard University, 1969.

2. Thomas Bayes, Essay Towards Solving a Problem in the Doctrine of Chances, The Philosophical Transactions, Vol. 53, pp. 370-418, 1763.

3. Bruno de Finetti, *Does it Make Sense to Speak of 'Good* Probability Appraisers'?, in The Scientist Speculates: An Anthology of Partly-Baked Ideas, I. J. Good, ed., pp. 357-364, Basic Books, New York, 1962.

4. W. R. Downs and C. Kasparian, A Computerized Method for Synthesizing Repair Time, McDonnell Douglas Astronautics Company Paper Number WD 1455, *also printed in annals of* Reliability and Maintainability, Vol 10, 1971.

5. Ward Edwards, A Bibliography of Research on Behavioral Decision Processes to 1968, Memorandum Report No. 7, Human Performance Center, University of Michigan, January 1969.

6. W. Edwards, H. Lindman, and L. J. Savage, Bayesian Statistical Interference for Psychological Research, Psychological Review, Vol. 70, pp. 193-242, 1963.

7. Goldman and Slattery, Maintainability, John Wiley & Son, New York, 1964.

8. G. Grippo and R. M. De Milia, Verification of Quantitative Maintainability Requirements, ESD-TR-65-220, Air Force Systems Command L. G. Hanscom Field, Bedford, Massachusetts.

9. Irving John Good, The Estimation of Probabilities – An Essay on Modern Bayesian Methods, The M.I.T. Press, Cambridge, 1965.

10. Charles Jackson Grayson, Decisions under Uncertainty: Drilling Decisions by Oil and Gas Operators, Harvard University, *Division of Research, Graduate School of Business Administration*, Boston, 1961.

11. Ronald A. Howard, The Foundations of Decision Analysis, IEEE Transactions on Systems Science and Cybernetics, Vol. SSC-4, No. 3, September, 1968.

12. Henry E. Kyburg, Jr., and Howard E. Smokler, Studies in Subjective Probability, John Wiley and Sons, New York, 1964.

13. Chi-Yuan Lin and George J. Schick, *On-Line (Console-Aided)* Assessment of Prior Distributions for Reliability Problems, *Annals of Reliability and Maintainability*, Vol. 9, 1970.

14. R. D. Luce and P. Suppes, Preference, Utility, and Subjective Probability, in Handbook of Mathematical Psychology, Vol. 3, R. D. Luce, R. R. Bush, and E. Galanter, eds., pp. 249-410, John Wiley and Sons, New York, 1965.

15. C. B. Morrison and D. J. Davis, Allocating Time-to-Repair Distributions, *Annals* of Reliability and Maintainability, Vol 13, 1974.

16. D. G. Morrison, Critique of: 'Ranking Procedures and Subjective Probability Distributions', Management Science, Vol. 14, pp. B253-254, 1967.

17. J. W. Pratt, H. Raiffa, and R. Schlaifer, Introduction to *Statistical Decision Theory (preliminary edition), McGraw-Hill*, New York, 1965.

18. Howard Raiffa, Decision Analysis: Introductory Lectures on Choices Under *Uncertainty*, Addison-Wesley, Reading, Massachusetts, 1968.

19. Howard Raiffa and Robert Schlaifer, Applied Statistical Decision Theory, Harvard University, Division of Research, Graduate School of Business Administration, Boston, 1961.

20. Leonard J. Savage, The Foundations of Statistics, John Wiley and Sons, New York, 1954.

21. Leonard J. Savage, The Elicitation of Personal Probabilities and Expectations, Journal of the American Statistical Association, Vol. 66, pp. 783-801, 1971.

22. Robert Schlaifer, Analysis of Decisions Under Uncertainty, McGraw-Hill, New York, 1969.

23. Carl-Axel S. Stael von Holstein, Assessment and Evaluation of Subjective Probability Distributions, The Economic Research Institute at the Stockholm School of Economics, Stockholm, 1970.

24. Carl-Axel S. Stael von Holstein, Encoding Subjective Probabilities *for Decision Analysis: Practical and Experimental Experience, presented at the Interdisciplinary Colloquium on* Mathematics in the Behavioral Sciences, UCLA, April 21, 1972.

25. Carl-Axel S. Stael von Holstein, A Tutorial in Decision Analysis, Unpublished manuscript, Stanford Research Institute, April, 1972.

26. Robert L. Winkler, The Assessment of Prior Distributions in Bayesian Analysis, Journal of the American Statistical Association, Vol. 62, pp. 776-800, 1967.

27. Robert L. Winkler, The Quantification of Judgment: Some Methodological Suggestions, Journal of the American Statistical Association, Vol. 62, pp. 1105-1120, 1967.

28. Robert L. Winkler, Probabilistic Prediction: Some Experimental Results, Journal of the American Statistical Association, Vol. 66, pp. 675-685, 1971.

29. G.J. Schick and R.W. Wolverton, "An Analysis of Competing Software Reliability Models," IEEE Trans. Software Eng., vol. SE-4, pp. 104-120, March, 1978.

30. R.W. Wolverton, "The Cost of Developing Large Scale Software," IEEE Transection on Computers, June 1974.

31. L.H. Putnam and R.W. Wolverton, "Quantitative Management: Software Cost Estimating," Tutorial, IEEE Computer Societies 1st International Computer Software and Application Conference (COMPSAC 77), IEEE Catalogue Number EH0129-7, Chicago, Nov. 8 - 11, 1977.

32. B. Littlewood and J.L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," 1973 IEEE Symp. Computer Software Reliability, NY, April 30 - May 2, 1973, pp. 70-76.

33. A.N. Suker, "An Investigation of Software Reliability Models," Rome Air Development Center, Griffiss Air Force Base, NY, Aug., 1976 (preprint); presented at the 1977 Annu. Reliability and Maintainability Symp., Philadelphia, PA, Jan. 18-20, 1977.

34. A.L. Goel and K.Okumoto, "An Imperfect Degugging Model for Reliability and Other Quantitative Measures of Software Systems," Technical Report No. 78-1, Dept. of Industrial Engineering and Operations Research, Syracuse Univ., N.Y., April, 1978.

35. A.L.Goel and K. Okumoto, "Classical and Bayesian Inference for the Software Imperfect Debugging Model," Technical Report No. 78-2, Dept. of Industrial Engineering and Operations Research, Syracuse Univ., NY, April 1978.

36. A.L. Goel and K. Okumoto, "Availability Analysis of Software Systems Under Imperfect Maintenance," Technical Report 78-3, Dept. of Industrial Engineering and Operations Research, Syracuse Univ., NY , April 1978.

37. A.L. Goel and K. Okumoto, "Bayesian Software Correction Limit Policies," Technical Report 78-8 Dept. of Industrial Engineering and Operations Research, Syracuse Univ., NY, April 1978.

38. Z. Jelinski and P.B. Moranda, "Software Reliability Research," McDonnel Douglas Astronautics paper WD 1808, presented at the Conf. on Statistical Methods for the Evaluation of Computer System Performance, Brown Univ., Providence, RI, Nov. 1971; also Statistical Computer Performance Evaluation, W. Freiberger, Ed., New York: Academic, 1972.

39. J.D. Musa, "A Software Reliability Model," Proc. of Second Summer Software Eng. Workshop, Goddard Space Flight Center, Greenbelt, MD, pp. 35-47, Sept. 19, 1977.

40. W.L. Wagoner, "The Final Report on a Software Reliability Measurement Study," Technol. Div., The Aerospace Corp., El Segundo, CA, Aug. 1973.

41. J.D. Musa, "A Theory of Software Reliability and Its Application," IEEE Trans. Software Eng., Vol. SE-1, pp. 312-327, Sept. 1975.

42. B. Littlewood, "A Semi-Markov Model for Software Reliability with Failure Costs," Proc. of the Symp. on Comp. Software Eng., New York, April 20-22, 1976, pp. 281-300.

DESIGN PROCESS ANALYSIS MODELING - AN APPROACH for IMPROVING the SYSTEM DESIGN PROCESS

Barbara C. Stewart
*Honeywell Systems and Research Center

ABSTRACT

A new discipline for improving the design process of large complex systems has been proposed. The discipline consists of a modeling technique (Design Process Analysis Modeling) combined with a set of analytical procedures. The Design Process Analysis Model and procedures are described, and some applications to such areas as computer logic design and software chief programmer teams, are discussed. A number of potential benefits of use of the discipline are identified, including: 1) Verifying that both technical design goals and organizational objectives are being met by the design process; 2) Providing a common approach for evaluating the cost-effectiveness of an existing organizational design process; 3) Establishing a framework within which different organizations can evaluate and compare design methodologies, tools, and design automation techniques. Additional areas for research are suggested.

1. INTRODUCTION

As all types of systems become larger, more complex, and more dependent on the use of sophisticated, rapidly changing computing technology, the design processes for such systems become more complex, costly, and higher risk. Today new computers must be designed and built using other computers, and new software for one computer is designed using a larger computer containing yet more complex software. With labor costs increasing and digital technology costs decreasing, increased automation of the design process for large, complex systems is becoming a necessity rather than an option. The problem in design process automation, however, is that design processes within any given organization are extremely complex, unique, and not well understood; also there are no commonly accepted analysis disciplines to determine the cost/benefits of automating a design process. Thus in most cases the results of design process automation are less than optimal. (Example: the software design process, which always involves use of computers, is one of the least understood, high risk, and high cost, types of system design.) A commonly accepted design analysis discipline is needed. It should be capable of wide application in different organizational and technology environments and should improve the success of any design process automation project.

*This work was jointly supported by: U. S. Office of Naval Research (Contract N00014-75-C-0650), U. S. Naval Air Development Center, and Design Analysis Associates, Inc.

2. TOWARD A DESIGN PROCESS ANALYSIS DISCIPLINE

During an initial attempt to develop a design process analysis discipline [1],astudy of the design processes of various organizations and technology environments resulted in the following observations:

Observation 1: *The most unique and constraining* factor in each industrial design process appeared to be the organzation, structure, and management of that process.^{x}

Observation 2: In the industrial organization and management of each design process, a number of critical information flow factors existed which are also critical in the management of other types of industrial processes.

Observation 3: The most "successful" (where "success" is defined differently by different organizations) large system designs were those where the organization and management of the design process itself were evaluated in terms of their impact on the system to be designed, and where necessary changes in the design process were made prior to the start of the design.

These three observations pointed toward potential applicability of traditional information systems analysis disciplines and industrial dynamics models to sort out the organizational from the technical elements of the design process.

In terms of the traditional systems analyst's input-process-output model, the design process is the "black box" (Fig. 1) which transforms inputs (the system requirements**) into outputs (the final system blueprints**), given a set of limited resources, external constraints, available technology, and within the boundary of a particular organization.

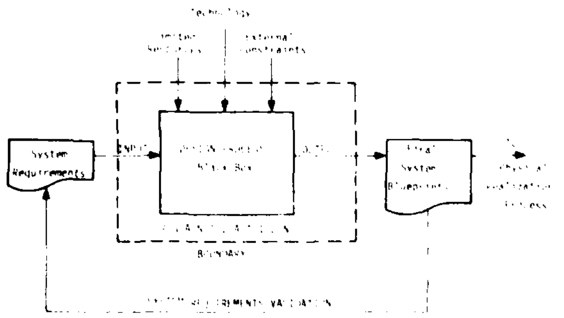


Figure 1. Design Process Modeled as Input-Process-Output

The problem can be stated as follows:

Design Process Analysis Problem:	How to analyze and specify the design process "black box" (Fig. 1) in such a way that the final system blueprints accurately and completely implement the system requirements, given technology, resources, external constraints, and within the the organizational environment.

Given Observations 1, 2, and 3 and the above problem statement,and given the results of several case studies [1] , we were led to the following assumption. It underlies the development of our design process analysis discipline:

Assumption:	De facto design processes already exist and are firmly enmeshed within the organizational structure, business goals, resource and technology limitations, and management style of particular organizations. New theoretical design methodologies and/or cost evaluation and prediction tools which require changes in the existing process will not achieve desired results, unless we first analyze the existing design process, its raison d'etre and its information requirements.

3. REQUIREMENTS FOR THE DISCIPLINE

Our concept of a design process analysis discipline involves a set of rules or procedures that can govern the analysis of any design process, and a top-level conceptual model of the process in which all of the key elements and their interrelationships can be identified. The intent of such analysis is "observation of the real world as it is ... development of best-fit models that describe but do not explain behavior, interpretation of the models, and the formulation and experimental confirmation, modification, or rejection of theories ..." [5]

The design process analysis discipline should provide a common framework for: a) explicit definition and study of the key parameters in a specific organization's design process, b) evaluation and comparison of the cost/benefits of various design methodologies and tools, c) comparison of design processes of different organizations and for differing technologies, and d) a starting point for more formal modeling, analysis, and prediction of design cost and risks.

Based on our initial work in this area[1] , a preliminary design process analysis model which meets these requirements has been developed.

4. THE DESIGN PROCESS ANALYSIS MODEL

The Design Process Analysis Model is an adaptation of Forrester's single-loop feedback system model [6]. This model provides a framework which captures all the critical elements of the design process and their interrelationships. The elemental single loop model (Fig. 2) consists of two processes, the Decision Process (a), as modeled by Gorry and Morton [7] , and the Design Change Process (b), wherein the most current system blueprint is modified by the set of design actions (c) initiated as output of the most recent decision. Design changes are embodied in design information (d) which is fed back as input into the decision process (a). Decision information (e) is also fed back for input to subsequent decisions. The entire single-loop design process is initiated externally by the set of goals, resources, requirements, and constraints (f) analysis representing the output of a higher level decision which initialize the decision process, and which are used to determine loop termination. (Termination occurs at the point when the design and design information indicates that the requirements and goals for that loop have been met. Each iteration of the loop takes a measurable amount of time and consumes a measurable amount of resources.)

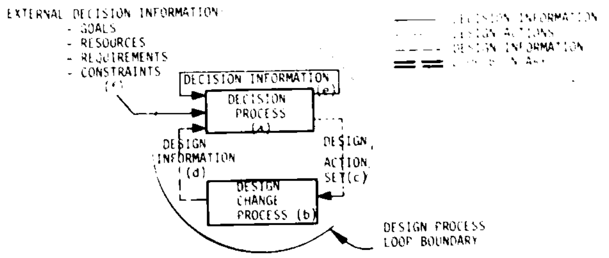


Figure 2. Single-loop Design Process Analysis Model Characterizes Critical Elements of Design Process and their Interrelationships

The model distinguishes among three different types of design process information: 1) Design (or "blueprint") information (d) which embodies the state of the system design at a particular time. Examples are engineering drawings, program listings, flow charts, HIPO diagrams, etc. 2) Decision information (e,f) which embodies all of the management and control information, requirements and constraints, criteria for acceptance of the design, and decision history. Examples are project cost data, design specifications, design goals, organizational objectives, project plans, progress reports, etc. 3) Actions (c) implemented by action languages and media which effect changes to the design (b).

Design processes for large, complex systems involving multiple organizational elements can be modeled as a multiple-loop design process where individual loops are interconnected by actions or information as shown in Fig. 3 (Detailed discussions of interconnection and modeling of multi-loop design processes will be found in [1]).

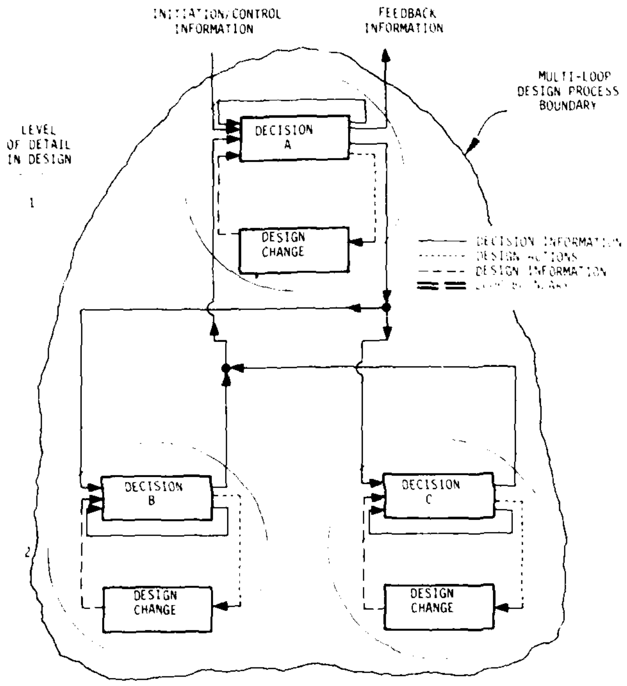


Figure 3. Example Multi-Loop Design Process Model Uses Decision Information Paths to Differentiate Among Levels of Design Detail and to Manage Design Process

5. SOME BENEFICIAL APPLICATIONS OF THE MODEL

The Design Process Analysis Modeling technique (Figs. 2 and 3) has proved useful in illustrating to non-technical managers the differences among various methodologies and design approaches. The following are some simple examples: a) In structured programming the limitation to three basic logic elements (sequence, if-then-else, and do-while) can be described in terms of the model as limiting the set of design actions (Fig. 2c); b) The differences between "top-down design" and "bottom-up design" can be expressed in terms of changes in the arrangement and connectivity of the design feedback loops; in terms of differences in the goals, constraints, and requirements at each design level; and in terms of changes in the design and decision information paths; c) Use of HIPO or SADT diagrams would apply to the format of the design information (Fig. 2d);d) Chief programmer teams can be expressed as a particular method of interconnecting the team members' decision and design change processes via the decision and design information and action paths; e) A design walkthrough is a method for closing certain design loops.

The model has also proved useful in providing insights concerning improvements in the design processes of specific organizations.

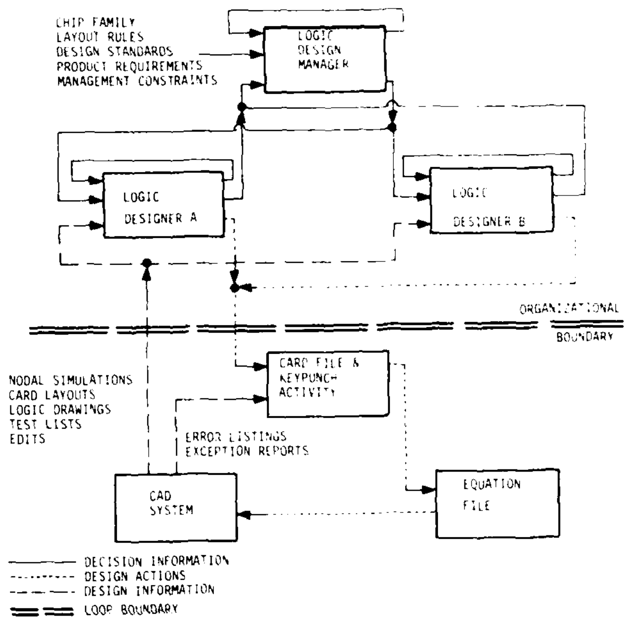


Figure 4. Computer Logic Design Process (Hypothetical Case, Highly Simplified)

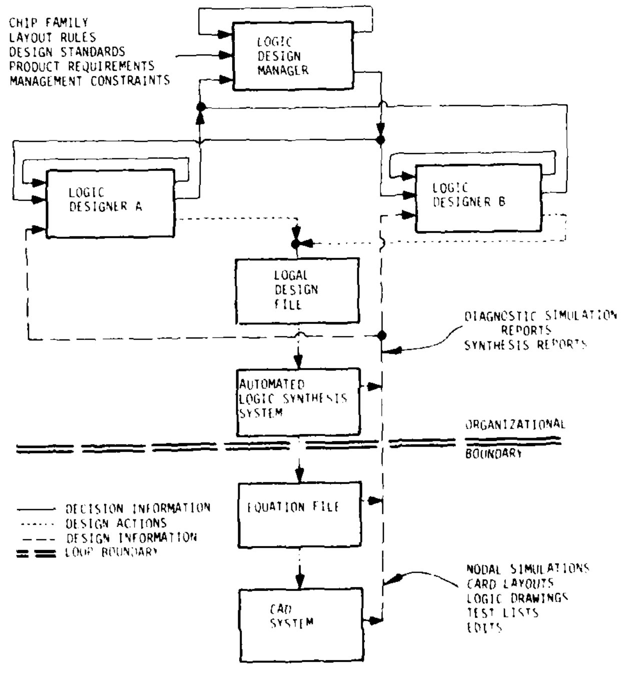


Figure 5. Revised Logic Design Process (after Design Analysis)

Fig. 4, for example, illustrates a highly oversimplified model of a *hypothetical design process* for a computer logic design group. Fig. 5 shows the same hypothetical process as modified by automation of key elements (which were identified through use of design process modeling and analysis)+. In the logic design example, major benefits resulted from closing the individual design loops earlier (by adding the logic simulation and synthesis system within the organizational boundary (Fig. 5). This, in the real world case, resulted in reducing the logic designers' turnaround time and the number of logic errors introduced into the equation file, which in turn reduced design cost.

Finally, the Design Process Analysis Model has proved useful in developing more detailed models and guidelines for certain types of design processes in systems which are reasonably alike and in organizational environments which are also reasonably alike. For example, a *model and guidelines have been* developed for the software development process of embedded computer systems, via analysis of the design processes of forty-five separate product types in eight different industrial organizations.

6. DESIGN PROCESS ANALYSIS PROCEDURES

As the model has evolved through its application in various environments, a number of procedures have been developed to systematize the design analyst's approach. These procedures are similar to those used in traditional systems analysis for automation of information processing systems, except that the Design Process Analysis Model is used as a framework.

The analysis procedures to be followed by the design process analyst include:

a. *Analysis* of the existing and/or future design process environment.

b. Understanding and documentation of present design process and its requirements: cost, time, and information flow, expressed in both English and in graphical form (c.f. Figs. 4 and 5).

c. Determination of current design process inadequacies and problem areas.

d. Analysis of overall design process information flow, timing, content, format, as well as the desired acceptable error rate in proposed modifications or automation of the design process.

e. Analysis of computerized end products and human end products desired for the improved or automated design process.

f. Analysis of similar or interfacing processes and systems within the organization.

g. Determination of organizational implications *and special problem* considerations involved in any new or revised design process.

h. Identification and evaluation of all alternatives for modifying the existing design process, (design methodologies and approaches, automated design tools, languages, etc.) including requirements for all the elements of the design process model (decision process, *actions, design process*, design information, and decision information).

Consistent application of these procedures within the framework of the Design Process Analysis Model, creates an analysis discipline which is useful in improving the effectiveness of design processes for any environment.

7. PRELIMINARY CONCLUSIONS CONCERNING THE DESIGN PROCESS

Preliminary application of the model and analysis procedures to several design processes has led to the following conclusions which are currently being verified:

a. Technology and technical skill requirements aside, information is the critical element in the design process, in relation to both the eventual quality and cost of the system being designed. As the number of people and organizational elements involved in the design process increases, the criticality of the information also increases (c.f. F. Brooks observations in [8]), and therefore the usefulness of design process analysis disciplines also increases.

b. Many existing design processes operate vitually open-loop; that is, there is no verifiable link (until late in the process) between either the system design goals and their embodiment in the design blueprint, or the organizational management goals and their embodiment in the de facto design process. (i.e. the design feedback loop duration is very long). As the number of people and organizational elements involved in the design process increases, the payoffs for establishing shorter design feedback loops also increase (reference TRW's experience [9]). Application of design process analysis techniques prior to the start of the design can be used to identify potential problems associated with lengthy design feedback loops, and to indicate where and how these loops might be shortened. (cf. differences between Figs. 4 and 5).

c. Each organization and design environment has an extensive and specific set of design and decision information processing requirements which are unique to that organization and its particular design process. Methodologies, techniques, and tools aimed at improving the design process should be evaluated in terms of the specific organizational design environment, in order to be applied effectively. Design analysis techniques provide a framework within which to compare design methodologies and tools, and to establish requirements for a particular design context.

d. Evaluation, analysis, and prediction of critical factors (such as design cost) become increasingly difficult as the number of people and organizational elements involved in the design process increases. Use of design process analysis provides a means for effectively sorting out the management, organizational, technical, and human factors and their contributions to overall design costs.

8. RECOMMENDATIONS FOR FURTHER STUDY

Based on our experience we have identified three major areas for further study in Design Process Analysis:

8.1 Refinement and Formalization of the Design Process Modeling Techniques and Analysis Procedures.

To date, the Design Process Model and Analysis Procedures have been used informally to help capture information concerning limited real-world design environments. In order to facilitate more widespread general use of these techniques, work needs to be done to make the models and procedures more formal, and to develop automated tools to aid in analysis and modeling.

8.2 Extension of the Verification Base

The model and procedures have been used to analyze only two classes of design processes, computer hardware design and embedded computer software design. We need to extend our application of the Design Process Analysis Model and procedures to additional classes of design processes in order to further verify our preliminary observations and conclusions.

8.3 Model Development as Framework for Evaluating and Comparing Design Methodologies and Cost Estimation Techniques.

A number of design methodologies and design cost estimation techniques are being developed. Using the Design Process Model as a framework, it would be useful to develop a standard approach (applicable to various organization and design environments) for determining the requirements for design methodologies and cost estimating techniques, and for comparing them.

9. SUMMARY

In this paper we have summarized some recent work in the development of a Design Process Analysis discipline, and have described a few of the benefits observed in our preliminary application of the discipline. We have also discussed some of our observations and conclusions drawn from experience with a Design Process Analysis Model. A number of recommendations for further research have been made.

ACKNOWLEDGMENT

The author wishes to express her gratitude to Professor G. Estrin of UCLA for his valuable discussions and invaluable support; to W. J. Dejka of the U. S. Naval Ocean Systems Center for his stimulating ideas on the "Science of Design;" and to J. H. Stewart for providing the information concerning the computer logic design process.

REFERENCES

** For the purposes of this paper, a "system" is loosely defined as a grouping of parts that operate together to achieve an explicitly stated common purpose (known as the "system requirements"), and the "final system blueprints" are defined as a one-for-one representation on paper of the final physically implemented system. (We here assume perfect mapping of final system blueprints to final physical system.)

+ Elements added are the use of LOGAL, a computer hardware description language, and a logic synthesis simulator which uses the LOGAL description as input.

x This observation is consistent with those of Lehman [2], Belady [3], and Kolence [4].

[1] B. C. Stewart: Design Analysis, An Approach for Improving the System Design Process, UCLA Computer Science Dept., Los Angeles, Ca., October 1976.
[2] M. M. Lehman: Evolution Dynamics - A Phenomenology of Software Maintenance, Software Life Cycle Management Workshop, U. S. Army Computer Systems Command, PP 313-323.
[3] L. A. Belady and M. M. Lehman, A Model of Large Program Development, IBM Systems Journal, Vol. 15, No. 3, 1976.
[4] K. W. Kolence, On the Relationships Between Design Theory and Software Life Cycle Management, Software Life Cycle Management Workshop, U. S. Army Computer Systems Command, PP 175-186, August 1977.
[5] M. M. Lehman, Software Life Cycle Management Workshop - Technical Introduction, Software Life Cycle Management Workshop, U. S. Army Computer Systems Command, PP3, 1977.
[6] J. W. Forrester, Principles of Systems, Wright-Allen, Cambridge, Massachusetts, 1968.
[7] G. A. Gorry and M. S. Morton, Management Decision Systems: A Framework for Management Information Systems, MIT Sloan School of Management, Cambridge, Ma. 1972
[8] F. P. Brooks, Excerpts from the Mythical Man Month, Datamation, December 1974.
[9] E. A. Goldberg, Applying Software Development Policies, AIAA Software Management Conference, Los Angeles, California, December 1977.

LIFE-CYCLE COST ANALYSIS OF INSTRUCTION-SET ARCHITECTURE STANDARDIZATION FOR MILITARY COMPUTER-BASED SYSTEMS

Harold Stone
University of Massachusetts
School of Engineering
Amherst, Massachusetts 01003

and

Aaron Coleman
U.S. Army
CORADCOM
Ft. Monmouth, New Jersey 07703

Abstract

This is a report of a life-cycle cost model to measure the effects of standardization of computer instruction-set architectures on military computer-based systems. The study considers six different scenarios, one of which assumes that standardization is not done, but only four different computer architectures are used. The remaining five scenarios consider the effects of standardizing on each of the architectures UYK-7, UYK-19, UYK-20, GYK-12, and UYK-41 (PDP-11).

Standardization impacts life-cycle costs in several ways. There is an inherent difference in the value and utility of the existing support software bases for the several architectures. The commercially supported architectures will augment and maintain a substantial portion of the softwar se free of government expenditure. Finally, some ar- chitectures are more efficient than others, and result in lower hardware costs if used as a standard.

The cost model attempts to incorporate these factors in a meaningful way to judge the relative importance of these factors and other factors on total life-cycle cost. The results of the model show that the GYK-41 results in the least life-cycle cost of any scenario over a broad range of annual rates of investment in support software. These conclusions are attributed to the fact that the GYK-41 ranks best or near best in each aspect that impacts total life-cycle cost.

Introduction

The objective of the study reported here is to measure the economic effects of standardization of computer instruction-set architectures on military computer-based systems. For the purposes of this study, the term <u>architecture</u> refers to the characteristics of a computer defined by its instruction repertoire. Two computers are said to have the same architecture if every assembly language program for one computer runs on the other and conversely. Two such computers may be vastly different in implementation and have radically different costs and performances. The commercial computer world has demonstrated that a family of implementations of a single architecture is feasible and desirable.

Military computer systems are starting to take on characteristics of commercial families in that prior-generation computers are being reimplemented with new hardware to take advantage of the technological improvements in cost and performance. At issue is the question of whether to standardize on a family of implementations of single architecture or to use a mix of many architectures, each used in an environment best suited to it. If standardization appears to be attractive, than a further question is which computer architecture should be used as a standard. Standardization can realize potential savings by eliminating duplicate efforts, but on the other hand it can incur additional costs if a standard is used in an environment for which it is not well-suited. The question then becomes one of determining which standard is the best overall standard.

The method used in this study to recommend a course of action is to compute the relative life-cycle cost for 78 representative Army/Navy computer-based systems which are acquired and deployed over a 22-year interval. The systems are acquired in lots of 26 for three different time periods--1980, 1985, and 1990--with each lot deployed for 10 years. R&D costs prior to each acquisition are included in the cost model. Thus the cost model serves to identify how a standard computer architecture can impact life-cycle costs, and gives some indication of the potential benefits and costs of the possible decisions. Any model of this type is subject to errors in estimates, so that the absolute dollar figures computed must be viewed as indicative of possible results rather than as predictions of the future. The model is successful in identifying the important factors, and in estimating their relative importance if in fact the model cannot be expected to predict dollar costs with absolute accuracy.

To isolate the key variable, computer architecture, from different hardware implementations, all computer systems are presumed to use the same family of modules and chassis in their implementation. These modules are presumed to be Military Computer Family (MCF) modules as specified by ITEK Corporation under contract to the U.S. Army. The modules use a common collection of memory modules, input/output modules and bus interfaces, with different CPU modules available to implement different instruction-set architectures. We presume that these modules can implement any of the instruction sets for the UYK-7, UYK-19, UYK-20, GYK-12, and UYK-41 (PDP-11) computers. The most likely methodology for realizing the collection of instruction sets from a common set of modules is to use CPU modules specific to each architecture that interface to the memory and input/output modules over a general bus.

The cost model identifies costs arising from principle sources (a) common costs, which are costs such as product planning, R&D, and support software required to mount an architecture in the field apart from costs for specific systems; (b) hardware life-cycle costs, which include acquisition, logistics, and maintenance costs; and (c) software life-cycle costs, which include initial acquisition operations, and maintenance costs.

A crucial problem in making the study is to assess the effects of the existing support software tools and the effects of continued commercial investment in architectures for which there are commercial counterparts. To this end we modeled the effects as follows:

1. A fixed annual government expenditure for support software is assumed to occur over the life-cycle. The study treats costs as a function of this level of investment.

2. Part of the investment is used to maintain the existing software base that is owned by the government. What remains after maintenance is used to procure additional support software.

3. Commercial tools incur no charges against government funds for maintenance.

4. Although it is likely that commercial investment may augment the support software base for commercial architectures, this model takes the conservative view that the present commercial base remains fixed at its initial value for the entire life-cycle.

This model assumes that

5. the greater the value of the software tool base, the lower the cost per line of code of applications software. An equation derived from actual cost data is used to predict this cost, and the model estimates the value of the tool base as a function of time to come up with time varying estimates of productivity.

The results of the model show that at all levels of software investment, the GYK-41 leads to the lowest life cost. The savings is $1.5 billion of the cost of a standard family of UYK-19 computers (22%), and about $5.1 billion (49%) of the cost of standard UYK-7 and GYK-12 families for an annual software investment of $2 million. Regardless how the specific effects of crucial variables are on total life-cycle cost, the GYK-41 will show up most favorably in comparison to the other architectures because it is at least as good or better than the other architectures in each of three key variables that impact life-cycle cost.

USEFUL EVALUATION TOOLS IN THE DESIGN PROCESS

C. E. Velez

Martin Marietta Aerospace
Denver Division, P. O. Box 179
Denver, Colorado 80201

Abstract

An overview of our research activities in the area of software design tools is presented. Keyed on the concepts of discipline, formalism and practicality of computer aided design processes, and the need for quality assessments at all stages in the life cycle, a concept of an integrated set of development and management tools is presented. Fundamental to this concept is the availability of a "static" quality measure applicable to a spectrum of specification levels of abstraction and the use of simulation as an integral part of the design process for dynamic analysis and validation.

Introduction

In considering the life cycle of a software system, various stages or milestones have been defined (e.g., AFR-800-14), ranging from requirements definition and analysis to installation maintenance. An awareness of total life cycle costs associated with software has surfaced a need for improvements in the techniques governing the earlier software development stages: requirements identification, analysis, and design development and validation applied through a continuum of abstraction levels leading to coding specifications. Key objectives of such techniques include identification of "optimal" modularization strategies, the surfacing of inconsistent, incomplete, or ill-defined requirements, and the validation of early products in the life cycle. An emerging vehicle for the accomplishment of these objectives are languages for identifying requirements, design components, or design specifications. These are collectively referred to here as design languages. Embedded around supporting software for language processing, design data base creation, maintenance, and assessment, such schemes become a design system which can serve as powerful tools for:

- A disciplined way of expressing designs,
- A distinct phase-to-phase staging of each design task with subsequent traceability of requirements,
- Computer-assisted documentation production as an automatic by-product,
- Management control facilities due to the high degree of visibility available through design data bases.

Several design languages that apply at different levels of the design process, result in different, concrete expressions of a design product. It is these expressions of the design product, stored as computerized data bases, which provide the key to a maximally controlled software development activity. Indeed the state-of-the-art in software engineering in general focuses on design data bases for some level of software expression.

The availability of a machine readable data base containing design information opens a whole field of possibilities for learning more about the design product. Going far beyond code analyzers (which use source code statements as the design level data base), we now have the means for a broad range of feedback generators to aid the human in iterative refinement activities.

Martin Marietta's expertise in this area is strengthened by a vigorous software engineering research program which continues to expand the state-of-the-art. Specific recent accomplishments have been achieved in the areas of design quality assessment as well as very high level languages for requirements and design (14), (16). Very high level languages allow an expression of a target system, i.e., a description at some level, which can be used to form a database. This database then provides us with a tangible, measureable object. (The major related work is with static/dynamic code analyzers to determine aspects such as program structuredness or complexity. This, however, is after-the-fact: design measures are more cost effective if they can be used before code is produced). Therefore, our current software engineering research program looks specifically at languages and features tailored to the measurement goal.

In order to use measurement as a mechanism for design assessment, some criterion is needed. That is, while we have the raw data for measurement, we still need to know what characteristic our "measure" will be informative of. One criterion that can be used is one of structural quality where "quality" can be associated with aspects of maintainability, reliability, testability, or modularity. The assessment scheme then produces a "static" design quality metric or index.

Before going further, we emphasize the distinction between software design as a process, and the design as an end-result or product. Of the many techniques proposed to positively influence the design process, they all are motivated by the common goals of producing a system that is structurally simple, maintainable, and testable, i.e., modules. If indeed, such characteristics can be induced into a design, the question arises as to how they would then be embodied. Our hypothesis is that the essence of such quality characteristics - and others as well - is representable by the structural aspect of the design. Furthermore, we believe that the structure which supports or "carries" these characteristics can be examined as to the extent that one or more design goals is met.

Another complimentary criterion is performance, which generally is measured by some level of simulation attacking aspects such as man-machine interaction, adequacy of computer hardware, etc. In the remainder of this paper, tools useful to support both these aspects of measurement will be briefly outlined.

Static Metrics and Decomposition

Design quality metrics have been addressed in the past from various viewpoints. Andreu (1) uses a strength and coupling measure applied to a graph representation of requirements and their interrelationships for preliminary design. Myers (9) has developed a model in terms of probability measures applied to discrete strength and coupling factors of program modules which is used to assess the ramifications of making program changes. Schutt, et al (15) have applied an information entropy measure to hypergraph representations of computer processes and data structures. And McCabe (7) shows a method for determining quality as a function of module "structured-ness". Each of these approaches relates in some way to a system decomposition measure. However, only Andreu directly addresses the problem of forming the decomposition itself. Parnas (11) also gives a strategy for decomposition (applied to the design process, rather than the product) but does not attempt a design metric even though considerable attention is given to resulting design quality.

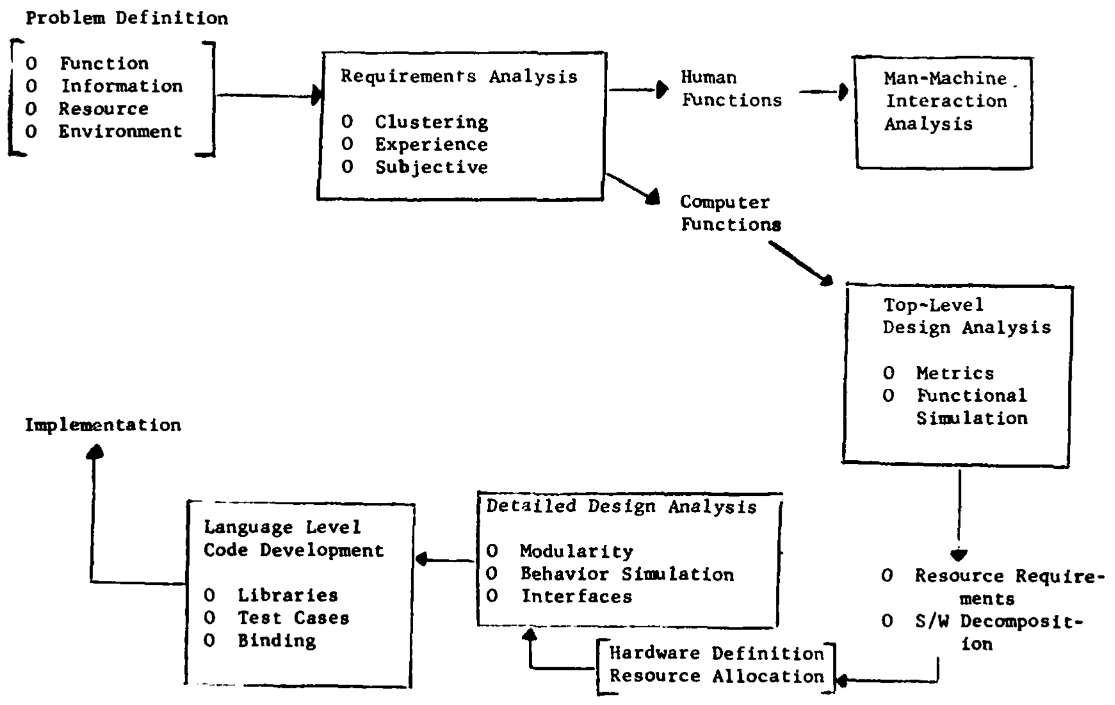
Each of these approaches assumes its own specific starting point located somewhere in the software development spectrum. One can think of all the points along such a spectrum as various levels of expression of the design. And expression, as a communication medium, will assume some linguistic form. These forms progress from natural language at the early concept/requirements stage to high order programming languages at the implementation stage. Rather than limit ourselves to a specific form or level of expression, we would like it to be arbitrary. In order to apply a design strategy and measurement scheme to a given expression, a transformation must produce an encoded form of expression to which methods for the analysis of structure and, eventually, quality can be applied. The transformation process establishes a graph by identifying and converting appropriate linguistic objects to nodes (vertices) and relationships to link (edges). Graph links between nodes must also be qualifiable as to strength, type, or importance; i.e., weighted.

The design process is equated to the decomposition of this graph into a collection of system "elements" (modules, programs, subsystems, etc.), i.e., a clustering activity, which can be subjected to standard metric analysis. Measures such as "strength" or "coupling" or "complexity" of a structure are generalized in this framework and used at varying levels of abstraction.

For a fixed class of problems we have determined how the choice of a metric will influence the quality measure produced - an adjacency matrix produces decomposition based on the number of interfaces between and within clusters. A distance matrix produces a similar result but is a more comprehensive criterion applicable to highly complex graphs. Other metrics which represent links as probability values, information channel quanta, or component control lines, would need further research to determine the characteristics of structure that quality indexes based on such metrics would reflect.

Simulation Tools

Our design approach includes a dynamic design validation at key stages in the design process. Two such stages are the definition of the man-machine interface (MMI) and ADP resource requirements (timing, memory, I/O parallelism, etc.) for the system. Key tools here include "virtual" I/O device simulation, functional simulation languages, emulations, etc. Research in this area has been directed at the interface definitions which allow such simulations to become a natural, cost-effective element of the design process. For example a top level clustering analysis might support a delineation between human operator and computer functions for an interactive application. In addition to design validation from the human factors point of view, a simulation driven by a requirements/design data base could surface key data/process structure implications and drive the test scenario development. Likewise, tools such as functional simulation of hardware and software can surface key resource allocation problems associated with the chosen hardware environment, if driven directly from the problem data base. Such interfaces form the basis for an integrated software design facility concept described in the attached figure.



Problem Definition
O Function
O Information
O Resource
O Environment
Requirements Analysis
O Clustering
O Experience
O Subjective
Human Functions
Man-Machine Interaction Analysis
Computer Functions
Top-Level Design Analysis
O Metrics
O Functional Simulation
O Resource Requirements
O S/W Decomposition
Hardware Definition
Resource Allocation
Detailed Design Analysis
O Modularity
O Behavior Simulation
O Interfaces
Language Level Code Development
O Libraries
O Test Cases
O Binding
Implementation

References:

1. Andreu, R. C., "A Systematic Approach to the Design of Complex Systems: An Application to DBMS Design and Evaluation," Center for Information Systems Research, Report # 32, MIT, 1977.

2. Andreu, R. C., "Set Decomposition: Cluster Analysis and Graph Decomposition Techniques," CISR Preliminary Report, MIT/Sloan School, June 1977.

3. Gileadi, A. N. and Ledgard, H. F., "On a Proposed Measure of Program Structure," SIGPLAN Notices, May 1974.

4. Hamilton, M. and Zeldin, S., "HOS - A Methodology for Defining Software," IEEE Transactions, SE-2, March 1976.

5. Hartigan, J., Clustering Algorithms, Wiley, 1975.

6. Jackson, M. A., Principles of Program Design, Academic Press, 1975.

7. McCabe, T. J., "A Complexity Measure," IEEE Trans SE-2, No. 4, December 1976.

8. Myers, G. J., "An Extension to the Cyclomatic Measure of Program Complexity," SIGPLAN Notices, October 1977.

9. Myers, G. J., Reliable Software Through Composite Design, Petrocelli/Charter, New York, N. Y., 1975.

10. Paige, M.R., "On Partitioning Program Graphs," IEEE Transactions, SE-3, No. 6, November 1977, p. 386.

11. Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," CACM 15, 12 (December 1972), p. 1053.

12. Robinson, L., "The Relationship of System Families to HDM" (Hierarchical Development Methodology) - Stanford Research Institute, TR: CSL-50, June 1977.

13. Roubine, O., "On the Design & Use of Specification Languages," SRI Technical Report CSL-48, October 1976 (AD/A-038 783).

14. Scheffer, P.A., "Computer-Aided Software Design," Martin-Marietta Internal Report D-22R, December 1977.

15. Schutt, D., "On a Hypergraph Oriented Measure for Applied Computer Science," CompCon Proceedings, September 1977, p. 295.

16. Velez, C. E., Scheffer, P. A., "On the Problem of Software Design and Measuring Quality", Martin Marietta Aerospace, May 1978, pp. 223-229.

PROGRAMMERS ARE TOO VALUABLE TO BE TRUSTED TO COMPUTERS

Gerald M. Weinberg

© Ethnotech, Inc. RFD #2 Lincoln, NE 68505

ABSTRACT

Most of the anticipated gains from programming tools have been slow in arriving, and often disappointing when they arrive. A major cause of this condition is the failure to understand the processes by which new technology is introduced. The role of training has been left by default to computers, under the assumption that they are cheaper or better than human teachers. Methods of learning without using either computers or teachers are not even considered, though they are far more economical and effective. The computer can have a role; the teacher can have a role; but unless the overall climate for professional learning is established, the computer and the teacher are hopelessly inadequate to the full job of creating a corps of professional programmers.

Choosing a Teaching Language

Every four years, along with county elections,the local computer science professors raise the question of the correct teaching language for programming. There's a lot of brave talk about throwing the rascals out, many lunches devoted to campaigning, a wave of confidence just before the election, and then the ultimate defeat of the upstart. In the end, both FORTRAN and the sheriff are reelected. They may be corrupt; they may be incompetent; they may be creaking with age; but they're at least familiar.

Like the county voters, professors are quite ready to rationalize the result. As the years since 1956 have accumulated, the points in these arguments have,one by one, withered away. And yet one remains, year after year, the backbone of conservatism everywhere. To quote a genuine professor:

> "My decision to base this course on the WATFIV programming language was founded not only on a recognition of real-world applications but also on the raw economics of computer costs at our installation."

"Real-world" and "raw economics" are nononsense words--none of your ivory-tower fol-de-rol. Let's look, therefore, at some of the "raw economics" of computer costs for training in the "real world".

Twenty years ago, computers were *really* expensive--so expensive that FORTRAN had to be ruled out of consideration for the language used in a programming course. In fact, even the richest of the rich thought using a *computer* in a programming course was a frivolous extravagance.

Twenty years is a long time in a sheriff's life, and even longer in the life of the programming profession. Few remember those days, or even *believe* that it was *possible* to teach programming without a computer. Soon, at the rate personal computers are spreading, few will remember what it was like to learn computing with a *teacher*. In the four years since that WATFIV decision was rationalized, the "raw economics" have changed so sharply that we could almost afford to give each student a personal computer. In this year of 1978, it sounds archaic to say "our installation" in the singular. What university worth accreditation hasn't got a dozen or more minis and micros scattered around the campus like Homecoming handbills?

Still, the FORTRAN argument has survived the latest reduction in hardware costs. Now it is the *micros* that can't afford to run anything else--except, heaven forbid, BASIC. Where does this conservatism originate? Where will it end? How far will it spread? Why does it resist the repeated efforts of language designers and implementors to break it down?

Using Software Tools

The programming language is merely the oldest and most familiar software tool, and universities are merely the oldest and most familiar tools for social change. The same conservatism in the adoption of new tools is found for all other tools in all other institutions. If we can answer our questions, the payoff could be staggering.

Of the many reasons for non-use of new tools, perhaps the most obvious is the lack of attention given to training. An elephantine sum of money has been spent on the development of software tools- the current rate probably exceeds a billion dollars a year. In contrast, a microorganismic sum of money has been spent on training people to use those tools.

Most of the tools--perhaps as a consequence of this disparity between development and training--are never or hardly ever used. Not only do people continue to use FORTRAN, but they continue to use it without, for instance, even getting cross-reference listings of their program variables. Even when their FORTRAN compiler provides such a cross-reference, the installation disables it, usually as a "standard", *because* "it costs too much". And, even where it is routinely produced, 90% of the programmers *never* look at it--yet the cross-reference listing is one of the simplest software tools, one of the most direct in its use, *one of the most* convenient, and one of the most ancient.

The situation is little better in the majority of installations that have abandoned FORTRAN for "higher" languages. As an exercise in formal review techniques, our clients and students study *published* programs, which presumably are held up as examples for novices to follow. In a typical review, of a program by two PhD Professors of Computer Science, we studied the use of PL/I. *Although the specifications* offered ideal situations for employing each of them, none of the following PL/I facilities were used:

1. dynamic allocation of storage
2. *cross-section notation*
3. array expressions
4. factoring of attributes
5. subscript expressions
6. control of type conversions
7. bit strings

One could almost have removed the semicolons and, in effect, compiled the program on a FORTRAN compiler.

Further examination of the program revealed, as is typical, *no impact* of the years of discussion of programming *style*. Among the more pitiful stylistic practices we found:

1. intricate branching, including into and out of loops

2. superfluous statements based on incomplete understanding of the action of earlier statements

3. initialization of variables upon *exit* from loops that used them

4. use of single-character names such as K and R

5. use and reuse of scalar variables in a program not pressed for storage

6. use of a name with a different meaning in the program and in a comment explaining its meaning

7. use of the keyword, PTR, as a data name in a most confusing context

8. general inconsistency in the naming of *variables*

9. computations inefficiently placed within loops, yet rendered inaccessible to an optimizer, in a program that heavily used pointers "for efficiency"

At the level of *design*, the program again showed no influence of recent discussion in the industry, let alone design tools and concepts. We found:

1. no checking whatsoever for valid input, either bounds or values

2. unchecked input used to control calculations, as in computed branches

3. a completely undesigned and error-prone input format

4. an algorithm which was inefficient for all but small cases of input

5. no monitoring of *performance of the* algorithm, which might have indicated loss of performance to the user

6. incomprehensible error messages

7. comprehensible error messages that were wrong or misleading

In our work, we have reviewed hundreds of programs from dozens of installations. The programs display approximately the same range of problems, and the installations display approximately the same non-use of tools. At least 75% of the installations routinely debug using unformatted hexadecimal dumps. At least 90% have never used a preprocessor. Program libraries are coming into use, but not in more than 50% of installations. Test data generation is rare; archiving of test data even more rare; and even a rudimentary data set comparison is hardly ever used.

We're *not* speaking of the number of these tools that sit on the shelf, accumulating dust and rent. We *are* counting the tool as "used" if *someone* uses it, even if for a minor part of its capabilities, so these figures overestimate real professional use. Since we are spending *some* money on training, *what* can it possibly be teaching them?

The Computer as Instructor

In view of the inadequate use of our expensive tools, it is obvious that we are not teaching programmers to use the computer adequately in their programming work. The conclusion seems obvious--we must spend *more* on computers in our classes, not less.

Obvious? It would seem so, until we look at the way the computer is being used in classes. Indeed, *we* are not teaching them at all--the computer is carrying the burden for us. As a result, we find ourselves standing on both sides of the same fence. It is the double thesis of this essay that

1. The computer is insufficiently used in programmer education.

2. The computer is far overused in programmer education.

Let's examine how the computer is used in a typical university course in programming. The nature of this use may best be understood in terms of an analogy. Suppose we have succeeded in developing a "paper-grader" program for high-school English courses, and that we have succeeded in getting the program used within the high-schools in the following way:

1. The teacher lectures on one topic or another to 50-500 students.

2. The teacher gives an assignment to write an essay.

3. The students write an essay, under strict orders not to help anyone else or to receive help from anyone else.

4. The essay is graded by our computer program on the basis of
   a. spelling errors
   b. grammatical errors

5. The paper is returned to the students with the grade.

What do you suppose the students will learn?

The reason this analogy is good is that we don't have to *guess* what the students will learn. We already *know*. Even without the computer, this is the way many high school English classes teach composition, and the results are notoriously bad. Many of the students learn to spell; some learn to avoid incorrect grammar; essentially none learn to *communicate*.

In programming education, the "paper grader" is already built into the compiler. Because programming assignments must be *recycled* through the grader until all "spelling and grammar" errors are eliminated, the emphasis on these aspects is all the stronger. In many classes, the professor has no time or stomach for reading the actual programs. In some, not even the *outputs* are read. In such classes, students turning in *wrong* outputs never find out from the instructor. Students turning in *fake* outputs *are* never found out *by* the instructor.

To solve the problem of unread output, sophisticated schools have developed "grader" programs which exercise the student programs using test inputs and scoring the resultant outputs. Graders definitely raise the level of computer assistance--but mostly to the harried instructor, not to the student. After all, for classes with hundreds of neophytes, how else can the legions of warm bodies be handled economically, or handled at all.

Grader programs, to give them their due, actually can represent an advance over the simple compiler checking that most schools still use. But, again, what is it *they* teach? By reviewing programs produced by these students, in class and years later, you can learn what the computer teaches about programming:

1. accurate card punching, in a batch environment

2. use of a text editor to overcome inaccurate keying, in an on-line environment

3. spelling of keywords

4. consistent spelling of programmer-chosen words

5. a subset of the syntax of a programming language

But beyond *these* important lessons, the computer--used in these ways--teaches a much deeper lesson, one that will remain long after the WATFIV syntax is forgotten. That lesson is:

Programs are shown to be correct by testing them.

This is a curious lesson. To quote the original sermon on structured programming by Edsger Dijkstra:

> "...the extent to which the program correctness can be established is not purely a function of the program's external specifications and behaviour but depends critically on its internal structure."

In short, what the computer is teaching is precisely antithetical to the principal lesson of the strongest movement for improved programming since the invention of the assembler. It is teaching this lesson every day, in every school in the country, to thousands and thousands of present and potential programmers.

But there is a second high-level lesson--a meta-lesson, actually--a lesson about learning itself:

> *We learn to program by throwing garbage into a computer and seeing what comes out.*

This lesson is like the prejudices of our youth--deeply set and hard to change. What's more, it's being taught earlier and deeper, now that personal computers are so readily available. What are we going to do with the *next* generation of programmers?

Render unto the computer...

Computers are excellent at teaching--about computers. For instance, no amount of lecturing about syntax errors seems to make the slightest impression on the majority of students--unless it is a slightly negative one. A compiler, though, patiently and mercilessly teaches syntax to one recalcitrant student after another. Cleverly used, it can even *motivate* some students to learn *principles* of syntax, though they have to obtain those principles elsewhere.

Some toolmakers say that syntax and spelling are unimportant lessons, because tools can be built to *correct* any errors. I can agree with them only partly, for no system will ever be able to correct *all* errors of any type. Consequently, even the most brilliant programmer must, at some time, learn the hard lessons of syntax and spelling--or else waste hundreds of frustrating hours.

Furthermore, even when the most sophisticated and careful proof techniques are applied to the most magnificently structured program, the computer may reveal two kinds of error that can slip through. First, there are the simple proofreading errors--errors that plague the most advanced mathematics journals as well as the most humble programs. Second, there is the complete misunderstanding of the problem.

Though a programmer may prove that the program does what she things it does, she can *never* prove that what she thinks it should do is what the user or users wanted it to do. In one sense, there are no wrong programs, only *different* programs. The only hope we have of discovering if we've solved the right problem is by giving the proposed solution back to the originator(s).

Consider the following utterly typical example. A professor of archaeology was teaching a large introductory course, assisted by the computer to the extent of its printing individualized examinations drawn at random from a pool of questions. An advanced student in the computer science department had been given the job of writing the program to print the exams, but the program had one slight flaw. Rather frequently, one exam contained the same question twice, or even three times. Rather than sampling the question pool "without replacement", the program was sampling "with replacement".

The professor noticed this defect and confronted the student. He was told that the program *could* sample without replacement, but that the process was "very inefficient". He knew that the professor certainly would not want to pay the extra cost, so it would be better to print, say 14 questions to be sure of getting 10 unique ones. The students just had to be told to answer "the first 10 unique questions".

The reader may want to speculate just which of the many poor algorithms this student programmer had chosen. In fact, the technique was conceptually very simple. When sampling "without replacement" was specified, the program would produce a tentative exam and then test to see if it contained any repeated questions. During program test, 10 questions were being drawn from a pool of 20, which meant that about 30 exams had to be drawn to get one without duplicates!

We must be careful to draw the correct moral from this tale. Most professors of computer science would lament this miserable student's ignorance of algorithms. Although that kind of ignorance is sad, we could learn to live with it in the "real world". What we can't live with is this student's ignorance of the programmer's role in life.

It was not for the programmer to decide how much the professor wanted to spend on exams. His guess about what the professor wanted led him to authorize printing of several thousand exams that had to be discarded. The deadline was missed, and the old system of a typed exam had to be used. Unless one is programming for one's own amusement, the final decision about whether or not a program is correct rests *not* with the computer, *not* with the programmer, but with the party with the problem. And woe unto us as the hordes of computer hobbyists hit the professional ranks--for them, *all* programming is strictly for personal amusement.

Another relevant point, of course, is that the programmer had not the slightest idea that his program was wrong. He didn't even know it was wrong from an "efficiency" point of view. This lesson, neither the computer nor the professor could teach.

There are, in the end, a multitude of lessons one must learn to become truly a professional programmer. Each such lesson has its own characteristic ways of being learned. In designing programming courses, we must see that each lesson is taught in the most *effective* way, and not merely that a certain number of students can be "processed" for a whole semester without demanding a tuition refund.

<u>How to Teach a Programming Course</u> <u>With or Without a Computer</u>

How do we design such a programming course? The general pattern should now be clear:

1. Use the computer to teach what only the computer *can* teach.

2. Use people to teach what only *people* can teach.

3. Make the "economic" choice between computer and people only in those cases where either can do the job with equal *effectiveness*.

No doubt an introductory course ought to begin with one or two encounters with the machine, to teach

1. the overall process by which programs get created

2. the finicky nature of computing machines

3. the mismatch between such machines and our abilities to be precise

After these lessons are taught, at least on an introductory level, the class should turn to its human resources to learn more difficult lessons. A typical assignment might involve:

1. a problem posed by the instructor acting as "user" but with certain lessons in mind

2. each student making a trial solution, on paper

3. each student evaluating the trial solution of another student

4. small groups of students, sometimes with instructor supervision, arriving at a composite solution

5. sometimes trying the composite on the machine

6. groups exchanging solutions for more formal review, sometimes guided by the instructor in front of the entire class

In such a class, the students ingest a much richer and more meaningful diet of programming information. They avoid time wasted on syntax and spelling, which are corrected as a byproduct of the informal evaluations--and which in any case could be taught by machine except for the fact that it's actually cheaper and more effective to do it through reviews! From student to student, from teacher to student, passes knowledge about *style*, about *language*, about *algorithms*, about *design*, about *tools*, about hundreds of little pieces of programming "widsom" that collectively set apart the professional from the amateur. And were it not too presumptuous, we might have mentioned that once in a while knowledge even passes from student to teacher.

But what of the "raw economics"? We've speculated about this. We've experimented with it. Our experiments surpass even our wildest speculations. Of course, the first saving comes because teams of 5 students run only one-fifth the number of assignments through the computer. Yet the lessons for each student are far greater than the old secretive method, for each student sees many approaches, not just one.

Secondly, the number of runs to produce a correct program (and not just a "working" program) is drastically reduced. The magnitude of reduction depends somewhat on the size of the problem, but a typical figure for student batch problems is from an average of 20 runs to an average of 2. Actually, *most* programs run correctly the first time on the machine--*and* the students can demonstrate it. They had better be able to; if not, the other teams cut it to pieces.

Thirdly, the programs themselves typically will run more efficiently--once attention is on design, rather than grammar and spelling. The factor of 30 lost by the archaeologist's programmer would not be untypical, but suppose we modestly put this factor as 2. Putting the three factors together we reduce machine costs by a factor of about 100 (5 x 10 x 2). Certainly that's enough "raw economics" to permit us to choose our programming tools on the basis of what they will teach, not what they will cost.

But the benefits to education do not stop with machine economics. We can use this method quite successfully when there is no compiler for the language we want to teach, when there is no compiler at all, and *even when there is no class*. When the probability of a program working the first time is so high, in many circumstances there is not much to be gained from actually running the program. Getting rid of the computer, or at the very least controlling the incredible variance it usually introduces, permits us to plan with more confidence, and to stick with the plan as the semester unfolds. Indeed, once freed of the constraints of the machine, the course economics depend mostly on the availability of teacher and classroom, but once beyond the barest introduction, *any* group of programmers can teach *themselves* in this way, on the job or off.

Actually, the economics of the on-the-job training reverses the usual college assumptions. The students are likely to be making higher salaries than the instructor, and certainly are when considered as a class. It simply doesn't pay to transport professional programmers to a cross-town campus for programming lectures when they can learn much more in less time by inspecting each others' work.

There is an implicit challenge here, to the professional teacher who wants to stay out of the unemployment lines. With the *right* leader, a small group of programmers can multiply their learning through review techniques. The leader has to help them invest the money saved on lectures and machine time in fruitful alternatives, such as instrumented runs that permit them to make design and algorithm comparisons that would be difficult to perform analytically.

The leader can give them more problems to do in the same time, or guide them in exploring more alternatives on the same problems. In this way, the students can be nudged along the path to design, rather than to even more obscure bit-twiddling. Tools can be evaluated in actual use, perhaps giving some return for the billions we've invested in them. But it will take a lot of running for an instructor to stay ahead in such an environment.

This new environment may not suit the old-fashioned teacher who sees the instructor job as a kind of Olivier playing Hamlet to packed houses of sleeping students. Neither may it suit the typical computer jockey whose narrow mind and dogged persistence have served to get so many A's from the usual programming classes. If such people are thus encouraged to leave the programming profession, it can be counted as another plus for this system of professional education.

Some Future History

If we study the development of other high technology fields, such as, electrical engineering, machine tools, telephonics steam power, and printing, we see that computing is not unique. We may be going through the stages faster, but we're going through them all the same. Our first quarter century has been fast-paced, and driven almost entirely by the "technological imperative". Hardware sales have been the alpha and omega of our narrow world. Anything that didn't promote the sale of more hardware was left behind in the rush for survival in the technology jungle.

For the past few years, since IBM "unbundled", the sales managers have come to understand that software is a product, just like hardware--but with an even greater potential market. The software rush is now in its adolescent stage, and repeating the entire hardware history at an even faster pace.

Yet, today, after an investment of perhaps 200 billion dollars in hardware and software, the achievements are relatively small. Relative to what? Relative to what the future--if history is any guide--holds in store. One measure of our immaturity is the lack of any true programming profession. The career path of choice for programmers today is "up and out". For most, there is nothing to learn after mastery of grammar and spelling, so there is little economic incentive to retain a higher-paid "experienced" programmer where a freshly trained beginner is available.

And, when an individual does surpass this narrow training, there is nobody in management to recognize how valuable such a professional really is. The managers, after all, once took a "programming" course. They know that programming is unprofessional, shallow, and unmanageable. They know that money spent on training is wasted, and would be better invested in some new hardware, or a software tool that promises to replace a few programmers.

All of have been hypnotized by a running sales pitch consisting of fallacious "raw economics" and illusory "real worlds". We have spent billions for "tools", but not pennies on understanding what is needed to create the professional technical leaders who will actually use them. We've spent millions on "schooling", but skimped on real learning. As a result, our tools lie on the shelf, misunderstood and little used. Our systems seem to cost too much, but conference after conference merely repeats the sales pitch of its predecessor--buy more *things*! Our systems fail to satisfy, but all we hear is that people don't understand the finicky nature of computers--the *next* generation will solve all that!

The next "generation" will come when we outgrow our adolescent fascination with *toys* and develop an adult interest in *people*. Then we will begin, as other technologies have done, to master the social and psychological forces that are the real power behind successful technology--and the real reason for technology in the first place. An excellent starting point would be to take computer training out of the hands of computers. And perhaps put it in the brains of people.

References

Much of what I know about how people use computers was summarized several years ago in

The Psychology of Computer Programming
New York: Van Nostrand Reinhold, 1971

Some good work has been done in isolated places since then, but has never been summarized in one place. Ben Shneiderman of the University of Maryland is working on such a summary, in the form of a book of readings, which I hope will soon be available.

Our own work since 1971 has been concentrated in five areas--structured programming; systems thinking; formal technical reviews; team programming; and programming tools. We've published several books and films on structured programming, seen as a human activity rather than a branch of mathematics or computer science. We've published several books on systems thinking and problem solving, and have more books and films in process. Our work on formal technical reviews is summarized in

EthnoTECHnical Review Handbook
Daniel P. Freedman and Gerald M. Weinberg
Lincoln, NE: Ethnotech, Inc., 1977

We are working on a similar handbook to summarize what we have learned about team programming. Our principal educational activity has been our Technical Leadership Workshop, in which we train professionals in all five of these areas, and in special courses centered on one area or the other.

Our principal published effort in the area of teaching and motivating people to use programming tools is

High Level COBOL Programming
Gerald M. Weinberg, Stephen E. Wright
Richard Kauffman and Martin A. Goetz
Cambridge, MA: Winthrop Publishers, 1977

We are now preparing a course for data processing management on the effective introduction of teams, tools, and technical reviews.

LIFE CYCLE MANAGEMENT MEASUREMENT
MODELS-PREDICTIVE

PROGRESS IN MODELING THE SOFTWARE LIFE CYCLE IN A PHENOMENOLOGICAL WAY TO OBTAIN ENGINEERING QUALITY ESTIMATES AND DYNAMIC CONTROL OF THE PROCESS

Lawrence H. Putnam

Quantitative Software Management, Inc.
1057 Waverley Way
McLean, VA 22101

Abstract

This paper reports on the progress made in dynamic phenomenological modeling of the software life cycle. An overview rationale is presented to demonstrate the need for a dynamic life cycle model. The software life cycle behaves like a narrow band gaussian process. The linkage to information theory is suggested and the concept that software systems have a characteristic bandwidth and behave like bandpass filters is used to quantify Brooks' Law and show why managers have little flexibility in specifying the development time of a system. Productivity varies inversely as the square root of the average applied manpower, hence managerial efforts to speed up projects, increase productivity and cut costs are non-productive because these measures are functionally related in a counterintuitive way. An example is presented to show how the Norden/Rayleigh model is used to generate quantitative answers to the management questions: Can I do it? How much will it cost? How long? How many people? What's the risk? What's the trade-off?

I. Introduction

To do meaningful tasks in the functional areas of modern government and business entities, most large scale software takes 2-3 years to develop, and has an operational life of 6 to 10 years before being replaced. This is a life cycle. A model of this life cycle has been developed and can be used to forecast and manage the costs, schedules and manloading requirements of large software projects. Time is the independent variable whether explicitly treated or not (careful attention to management thinking, guidance, etc., will show that time is always implicitly recognized and is central to their planning and thinking--all budgets, plans, and contracts have a time base).

Managers ask these questions of software projects just as they do of any other process that consumes resources:

How much will it cost?

How long will it take?

Because the answers to these questions will go into a resource allocation plan, or budget, a time base is implied and the manager is really asking for much more. Typically, he really wants these answers:

Can I do it?

How much will it cost?

How long will it take?

How many people at any time?

What kind of skills?

What are the risks?

Are there trade-offs? What are they?

How do the constraints affect these answers?

These are the questions we can now answer. In Section II a broad overview of the software cycle is provided. Section III shows how to apply this model to a real world software project. Answers to the management questions are determined quantitatively and presented in a form directly usable by financial and project managers. To provide a framework for this work, the fundamental aspects of software systems development and its interrelationship with physics and information theory is presented in Section IV. Finally, summary tables are presented in the last section for quick reference to the techniques necessary for forecasting costs and schedules.

II. Overview of the Software Life Cycle

Many industrial processes are linear or nearly linear. This means that

Quantity = rate x time

where rate is a constant.

In a people-intensive activity it means

Effort = manpower x time

Cost = Cost/unit effort x effort

where manpower (a rate) is assumed constant and cost/unit effort is a standard cost, or a time average.

A further assumption is that the production rate (number of "widgets"/day) is directly proportional to manpower (effort/day), or since the time base is the same,

Effort Product (no. of items)

This doesn't work for software because software production rates are not linear with time and the effort is not directly (linearly) proportional to product.

Sometimes mildly non-linear rates can be replaced by averages over time (standard labor costing is an example).

If the rates are continuously varying then the solution lies in the use of the calculus. This is a scary word because many people have not studied it and have heard that it is vague, abstract, hard to apply and only long hairs use it to prove obscure scientific points of no practical consequence.

We have to accept that built-in bias, solve the problem using calculus and then present the results in a tabular and graphical form which can relate the curvilinear relationships in a form palatable to most planners and decision makers.

Now, if our problem is not only curvilinear in its fundamental relationships but also possesses a random character, this means the probability and statistical laws come into play. It means we cannot measure rates, effort, time, and product with great precision because these quantities are always fluctuating about some average (and that average may be changing over time), so what we have to do is work with averages of the quantities and a measure of the variability of the quantities (standard deviation). This is an important philosophical point because it means that only a certain level of accuracy and precision is possible and all efforts to do better are futile. It further means that the statistics improve precision with higher and higher levels of aggregation and get worse with increasing levels of disaggregation. The implication of this is that precise, detailed, work breakdown structures are highly unlikely to be accurate or meaningful in a quantitative way.

Given this background, the software estimating and control problem is really a different problem depending on where one is in the software life cycle.

During the feasibility and functional design phases, it is a pure estimation problem using phenomenology and past experience (data) to forecast a time varying future event. A model of the observed behavior is appropriate. This model should be a time varying model and should have parameters that relate directly to the management questions. The Rayleigh/Norden model Putnam has chosen meets these criteria.

A life cycle forecast is an intelligent "guess" of how a process will behave over an 8 or 10 year period based on the information we have now. But our input information about the system we plan to build, or are building, is continually changing for a variety of reasons. Requirements change because of external factors (regulatory; the way we do business, etc.).

This requirements change process means that our estimate done a short time ago becomes obsolete and needs revision to reflect today's reality.

This can be done by just re-estimating using the new input information if the development has not yet started. However, if development is underway the problem is different. We have to adapt on the fly. We have to assess what an incremental change applied at a specific time will have on the process at all future times. Clearly, this is a dynamic modeling requirement. Again we want the answers expressed in the form of answers to the management questions so that managers can reprogram resource allocation and assess risks in terms of contractual obligations.

Now a static model that does not adapt to the real data coming from the actual project can only do this in a very imperfect and superficial way. All the current models (Doty, PRICE-S, IBM, GRC, TRW) except one (Putnam) are static - they do not treat time explicitly and they do not have the capability to adapt to the actual behavior of the system at any instant of time in that 8-10 year life cycle period.

The best data you will ever have is that coming from the project you are working on. A careful fitting of that data to a model that is faithful to the phenomenology involved will give the best possible answer and, moreover, will continually adapt to the dynamics of the change process. This means that it is always converging toward the true answer which, alas, is always unknown (exactly) until the process is over. But the convergent property is highly useful because it means we are always getting closer and closer to the true behavior even though we can only determine the average behavior and the statistical uncertainty (due to the random character) at any instant.

But the statistical uncertainty is useful. It lets us control the process. If we can estimate that one year from now a system should require 25 people with a statistical uncertainty ($\pm 1\sigma$) of 5 people, and we find when we get to that point that 40 people are working on the system, then higher management will know that something significant has happened due to a real cause because there is less than 1 chance in 100 that such a large excursion from the predicted would occur because of pure chance (random fluctuation).

This is academic for the 1st line project manager; he knew when he had to add people, or shift resources. It may not be apparent to financial and budgetary officials well removed from daily contact who review the project only periodically from records and reports.

On the other hand an excursion of ' 3 or 4 people would be normal - inherent in the random character of the process - no cause for concern.

So an adaptive model is necessary for dynamic update to tell us where we are and where we appear to be heading based on all information available now. The Rayleigh/Norden model does this by continually fitting each piece of new information to revise the most recent estimate. The technical term for this is "adaptive filtering". It is really a real time process controller that gives the optimal future resource allocation (consumption) pattern at any instant in time. The Box (13,15) fitting technique is the implementation of the adaptive filtering concept.

The requirements change process is modelled using the second order Rayleigh differential equation. This can be made to respond in real time because it can be solved step-wise in discrete fashion using the very general Runge-Kutta numerical solution to differential equations. The theory is a little deep, but the implementation is very straight forward on a calculator or computer because all one has to know is:

- the actual elapsed time, t
- the actual manpower on board, $\dot{y}$
- the cumulative effort expended so far, y
- the best estimate of the difficulty from previous experience, K/t_d^2, where K is the life cycle effort and t_d is the development time,
- and an estimate of how much the difficulty is likely to change as a result of a requirements change. Linear thinking is reasonably valid here. If about 25 of the application programs are affected by a requirements change then there will be about a +25 change in the difficulty.

So,

$$\ddot{y} + t/t_d^2 \, \dot{y} + 1/t_d^2 y = D\,(1 + .25)$$

$$= K/t_d^2 \, (1 + .25)$$

is the form we used to study the impact of the requirements change process. We just project ahead at $t + \Delta t$, $t + 2\Delta t$, $t + 3\Delta t$, etc., and see how $\dot{y}$, y compare with the earlier estimates made prior to the impact of the requirements change process. The difference will be the incremental manpower and effort required to accommodate the change.

Tracking and fitting throughout the life cycle is important because 60 of the life cycle effort goes on in the operations and maintenance phase. If this is treated as a level-of-effort task, then far more resources than are necessary are used. This inhibits future development capability given budget constraints. Moreover, some 70-80' of total software work is in the operations and maintenance phase so unless this work is optimally controlled, then the projection of Harlan Mills* will become true and there will be no capability to do new development work if the software house has to work with a fixed manpower or budget constraint (very common in government).

Given this background, let's examine how we use the Rayleigh/Norden equation to obtain engineering quality answers during the early specification and functional development phases of a software project. The following example will illustrate simple applications of several powerful techniques.

III. Example of an Early Sizing, Cost and Schedule Estimate for an Application Software System

Software development has been characterized by severe cost overruns, schedule slippages and an inability to size, cost and determine the development time early in the feasibility and functional design phases when investment decision must be made. Managers want answers to the following questions: Can I do it? How much will it cost? How long will it take? How many people? What's the risk? What's the trade-off? This portion of the paper shows how to size the project in source statements (S_s), how to relate the size to the management parameters [life cycle effort (K) and development time (t_d)] and the state-of-technology (C_k) being applied to the problem through the software equation, $S_s = C_k K^{1/3} t_d^{4/3}$. The software equation is then solved using a constraint relationship $K = |\nabla D| t_d^3$, where $|\nabla D|$ is the magnitude of the difficulty gradient empirically found to be related to system development characteristics measuring the degree of concurrency of major task accomplishment. Monte Carlo simulation is used to generate statistics on variability of the effort and development time. The standard deviations are used to make risk profiles. Finally, having the effort and development time parameters, the Rayleigh/Norden equation is used to generate the manpower and cash flow rate at any point in the life cycle. The results obtained demonstrate that engineering quality quantitative answers to the management questions can be obtained in time for effective management decision making.

Background and Approach

Over the past four years the author has studied the manpower vs time pattern of several hundred medium to large scale software development projects of different classes. These projects all exhibit a similar life cycle pattern of behavior - a rise in manpwer, a peaking and a tailing off. Many of these

*Mills projected saturation with maintenance work a few years ahead leaving no capability to do new work unless there was an ever expanding software work force. Mills implicit assumption was level-of-effort on existing systems and no death process. The Rayleigh equation automatically accounts for obsolescence and death and hence provides for new development capability. This is what actually happens and is demonstrated by the data.

projects (and all the large ones) follow a time pattern described by the life cycle curves of Norden (7,8) which are of the general Weibull class and more specifically the Rayleigh form,

$\dot{y} = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2}$, where $\dot{y}$ is the manpower at any time t; K is the area under the curve and is the nominal life cycle effort in manyears; t_d is the time of peak manpower in years and corresponds very closely to the development time for the system.

Even though large systems seem to follow this general pattern, some small systems do not. They seem to have a more rectangular manpower pattern. The reason for this is that the applied manpower pattern is determined by management and by contractual agreements. Many small projects are established as level-of-effort contracts - hence rectangular manloading. For large projects this is generally inadequate because managers have a poor intuitive feel for the resources to do the job. Accordingly, they tend to respond to the needs of the system reactively. This results in time lags and underapplication of effort at some instant in time, but the effect is a reasonably close approximation to Rayleigh manloading.

The author has shown in earlier works (9-14) that there is a Rayleigh law at work. It is the 1st subcycle of the overall development curve called the design and coding curve (detailed logic design and coding). This is also a manpower curve that is proportional to the analyst and programmer manpower - the direct productive manpower. This curve is denoted $\dot{y}_1$. Its form is

$\dot{y}_1 = K/t_d^2 \; t \; e^{-3t^2/t_d^2}$ (MY/YR) when related to the original definition of K and t_d for the overall burdened life cycle curve. When this curve is multiplied by the average productivity ($\overline{PR}$) for the project it yields the rate of code production.

$\frac{dS_s}{dt} = \dot{S}_s = 2.49 \; \overline{PR} \; \dot{y}_1$, where the 2.49 is necessary to account for the definition of productivity as a burdened number (i.e., includes overhead and support activities). Now the time integral of the rate of code production yields the total number of source statements,

$$S_s = \int_0^\infty \frac{dS}{dt} \, dt = \overline{PR} \; 2.49 \int_0^\infty \dot{y}_1 \, dt$$

$$S_s = \overline{PR} \, . \, 2.49 . K/6.$$

The author has found that the $\overline{PR}$ is related to the Rayleigh parameters K and t_d in the following manner (14):

$\overline{PR} = C_n \, (K/t_d^2)^{-2/3}$ where the term K/t_d^2 has been defined as the system difficulty in terms of effort (K) and time (t_d) to produce it and C_n is a quantized constant defining a family of such curves. C_n is a channel capacity measure in the information theory sense, but in a more practical sense, it seems to be a measure of the state-of-technology being applied to a particular class of system.

Substituting for $\overline{PR}$, we obtain the software equation:

$$S_s = 2.49 \; C_n (K/t_d^2)^{-2/3} \; K/6$$

$$S_s = \frac{2.49}{6} \; C_n \; K \; K^{-2/3} \; t_d^{4/3}$$

$S_s = C_K \; K^{1/3} \; t_d^{4/3}$, where C_K has now subsumed $\frac{2.49}{6} \; C_n$.

Having this expression which now relates the product in source statements to the Rayleigh manpower parameters (which are also the management parameters), we turn to a practical way in which to estimate the size (S_s), effort (K) and development time (t_d) of a software project early in the requirements and specification phase of the project. This will let us answer the management questions necessary for effective investment decisions for the software project.

We will do this in the form of a case history for a project we will call SAVE. First, we will show a way to obtain a good estimate of the number of source statements. We'll plot the software equation and establish a feasible region for our development time parameters, we will impose a constraint relation involving K and t_d. We will do a Monte Carlo simulation to generate variances for K and t_d. With these numbers in hand, we can then do a trade-off analysis, pick a reasonable effort (cost) time combination and complete our translation into quantitative answers to the management questions. The answers we obtained will be close to optimal for the given constraint and, moreover, we will automatically have a sensitivity and risk profile.

Initial Sizing

Given the broad, preliminary design of SAVE consisting of the processing flow of the major functions and the estimates by the designers of the size range of the major functions, we can make a preliminary estimate of the development time, development effort and development cost to build the system. (See Figure 1.)

The input data from the project team are in the form of size ranges for each major function. Three or four team members estimated the size of each function as follows:

- Smallest possible size (in source statements) - a
- Most likely size - m

APPLICATION SOFTWARE:

SOLVING THE SIZING-ESTIMATING PROBLEM

- **FEASIBILITY SIZING**
  - ESTABLISH BOUNDS ON SIZE (DELPHI, BAYESIAN INFERENCE, SIMULATION)

    ⬇
  - ESTABLISH BOUNDS ON EFFORT, \$, DEV. TIME ($\pm$ 75-100 %)

- **DECISION SIZING** ($\leq \pm$ 25% EFFORT, \$, t_d)
  - PRELIMINARY DESIGN DONE (KNOW MAJOR FUNCTIONS- S_s ESTIMATE, TECHNOLOGY STATE POSSIBLE)

- **CONVERGE TO TRUE BEHAVIOR**
  - FIT REAL DATA (ADAPTIVE FILTERING)

20-40% of software work

- **CONTROL**
  - FIT REAL DATA (MINOR PARAMETER ADJUSTMENT)

60 - 80 % of software work

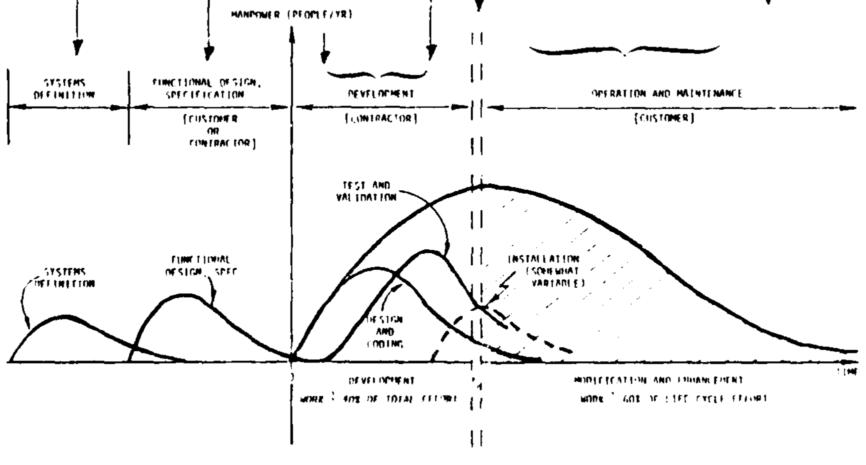


Figure 1. THE SOFTWARE LIFE CYCLE

- Largest possible size - b

These were averaged for each function and resulted in the first 3 columns of Table 1. This was in effect a Delphi polling of experts and their consensus. (Having done this with several groups of systems engineers, it is interesting to note that they are very comfortable with this procedure.)

Note that this results in a broad range of possible sizes for each function and that the distribution is skewed on the high side in most cases. This is typical of the Beta distribution, the characteristics of which are used in PERT estimating. We adopt the PERT technique to get an overall system size range and distribution.

1. An estimate of the expected value of a Beta distribution is:

$$E_i = \frac{a + 4m + b}{6}$$

The overall expected value is just the sum of the individual expected values.

$$E = \sum_{i=1}^{N} E_i .$$

This is the sum of the fourth column of Table 1 (98475 S_s).

2. An estimate of the standard deviation of any distribution (including Beta) is the range within which 99% of the values are likely to occur divided by 6, i.e.,

$$\sigma_i = |b-a| /6.$$

The overall standard deviation is the square root of the sum of the squares of the individual standard deviations, i.e.,

$$\sigma_{tot} = \left(\sum_{i=1}^{N} \sigma_i^2 \right)^{1/2}$$

This results in a much smaller standard deviation than one would "guess" by just looking at the individual ranges: the reason is that some actuals will be lower than expected (E_i); others will be higher. The effects of these variations tend to cancel each other to some extent. This cancelling effect is best represented by the root of the sum of squares criterion.

The result is

$\hat{E}$ = 98475 source statements

$\hat{\sigma}_{tot}$ = ± 7081 source statements

and the 99% range is 77,000 - 120,000 S_s, or we are 99% sure that the ultimate size will be in this range if the input estimates do not change. Of course, if the input estimates change, we should redo our calculations and revise the results accordingly.

Development Time-Effort Determination

Table 2 is a result of using the software equation which relates the product in source statements to the effort, development time and state-of-technology being applied to the project. The equation is derived partly from theory and partly from an empirical fit of a substantial body of productivity data. The form of the equation is:

$$S_s = C_k \ K^{1/3} \ t_d^{\ 4/3}$$

where S_s is the number of end product delivered source lines of code, an information measure.

C_k is a state-of-technology constant. For the environment anticipated for SAVE this constant is 10040. C_k can be determined by calibration against the software equation using data from projects developed by the same software house using similar technology and methods.

K is the life cycle effort in man years. This is directly proportional to development effort (Dev Effort = .4K) and cost ($\$/\overline{MY}$. K = \$LC cost; $\overline{\$/MY}$. (.4K) = \$ Dev).

t_d is the development time in years. This corresponds very closely to customer turnover.

Figure 2 shows a parametric graph of this equation.

Table 2 presents three scenarios for 5 different points in the size distribution curve. The expected case is given in the row labelled E. The column under t_d = 2 years gives a nominal development effort of 23.59 man years, \$1.18M cost (@ \$50,000/MY) to do 98475 source statements.

The fastest (or minimum) possible time for 98475 source statements is 1.81 years. The corresponding development effort is 35.4 MY, and cost of \$1.77 million. The assumption here is that the system is a stand alone and the gradient condition of |VD| = 15 cannot be exceeded.

The risk biased column is based on deliberately adding time (.4 of a year) to the minimum time to increase the probability of being able to deliver the product at the contract specified date. This biasing is to allow for external factors such as late delivery of a computer, an average number of requirements changes during development, etc. In the case of 98,475 source statements, this would be 1.81 + .4 = 2.21 years. The corresponding expected development effort is 15.91 MY; \$.8 million cost. Note that development effort and cost go down as time to do the job is increased. This is Brooks' law at play. Conversely, there is no free lunch--

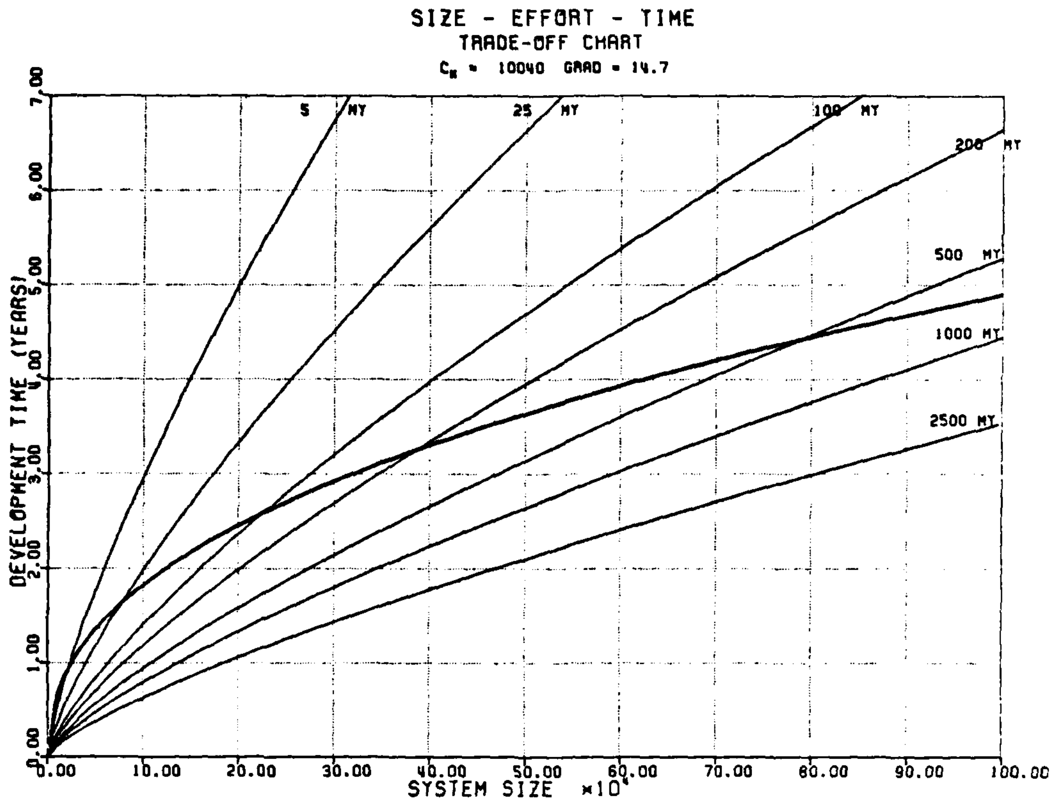

SIZE - EFFORT - TIME
TRADE-OFF CHART
C_K = 10040 GRAD = 14.7
5 MY
25 MY
100 MY
200 MY
500 MY
1000 MY
2500 MY
DEVELOPMENT TIME (YEARS)
SYSTEM SIZE ×10⁴
0.00
1.00
2.00
3.00
4.00
5.00
6.00
7.00
10.00
20.00
30.00
40.00
50.00
60.00
70.00
80.00
90.00
100.00

Figure 2

Table 1.

Major Function	S_s Least a	S_s Most Likely m	S_s Most b	Expected S_s	Standard Deviation S_s
Maintain	8675	13375	18625	13467	1658
Search	5577	8988	13125	9109	1258
Route	3160	3892	8800	4588	940
Status	850	1425	2925	1579	346
Browse	1875	4052	8250	4389	1063
Print	1437	2455	6125	2897	781
User Aids	6875	10625	16250	10938	1563
Incoming Msg	5830	8962	17750	9905	1987
Sys Monitor	9375	14625	28000	15979	3104
Sys Mgt	6300	13700	36250	16225	4992
Comm Proc.	5875	8975	14625	9400	1458
				98475	7081

Table 2. SAVE

Assumption: On-line, interactive development, Top down structured programming, HOL, Contemporary development environment. No machine constraints.

C_k = 10040; Standalone System -- |[illegible]| = 15

	S_s	t_d = 2 yrs	Fastest		Risk Biased	
		Dev Effort (MY)	t_d	Dev Effort (MY)	t_d + .4 yr	Dev Effort (MY)
-3σ	77000	11.28 (\$.564M)	1.63	25.80 (\$1.29M)	2.03	10.71 (\$.55M)
-1σ	91394	18.86 (\$.943)	1.75	32.16 (\$1.61)	2.15	14.12 (\$.71M)
E	98475	23.59 (\$1.180M)	1.81	35.40 (\$1.77M)	2.21	15.91 (\$.796M)
+1σ	105556	29.05 (\$1.45M)	1.86	38.71 (\$1.84M)	2.26	17.77 (\$.89M)
+3σ	120000	42.69 (\$2.135M)	1.97	45.65 (\$2.28M)	2.37	21.77 (\$1.09M)

if time is shortened the cost goes up, dramatically.

This can be illustrated by obtaining the trade-off law from the software equation. Solve the software equation for K:

$$S_s = C_k \; K^{1/3} \; t_d^{4/3} = C_k \; (Kt_d^4)^{1/3}$$

$$Kt_d^4 = \left(\frac{S_s}{C_k}\right)^3$$

$$K = \left(\frac{S_s}{C_k}\right)^3 / \; t_d^4.$$

This is the trade-off law. In terms of development effort, E = .4K so

$$E = .4\left(\frac{S_s}{C_k}\right)^3 / \; t_d^4 \text{ MY}$$

In our specific case

$$E = .4\left(\frac{98475}{10040}\right)^3 / t_d^4$$

and we can trade-off between 2 years (contract constraint, say) and 1.81 years--the minimum time for our gradient constraint.

Parameter Determination By Simulation

While Table 2 gives a fairly broad range of solutions that answer many "what if" questions, it is an essentially deterministic solution; that is, it assumes we know the input information exactly. Of course, we don't.

A better solution, then, is one in which we treat the uncertainties in our input information in obtaining our solution. This is generally not feasible analytically, but is nicely handled by Monte Carlo simulation. In our case we do this by letting the input number of S_s vary randomly about the expected value (98,475) according to our computed standard deviation, σ_{S_s} = 7081, and letting the stand-alone gradient ($|\nabla D|$ = 15) vary within the statistical uncertainty of its measured (computed) value ($\sigma_{\nabla D}$ = 2).

We then run the problem on the computer several thousand times with these random variations in parameters and generate the statistics of the variation in our answer. This is a much better measure of what is likely to happen as a result of the uncertainties in the problem.

The results of the simulation are given in the next table. Notice that the simulated estimated development effort is the same as the expected deterministic value and the development time is also the same. This is as it should be. The simulation produces the right expected values. The real value in the simulation is that it produces a measure of the variation in effort and in development time which we can used to construct risk profiles.

Major Milestone Determination

The results of the simulation determination of the development time are used to generate the major milestones of the project.

These milestones relate to the coupling of sub-cycles of the life cycle to the overall project curve (9-14). Examination of several hundred systems shows this coupling is very stable and predictable. The empirical milestones resulting from these earlier studies shows the following scaling.

Event	Milestone Fraction of Development Time, t_d
Critical Design Review	.43
Systems Integration Test	.67
Prototype Test	.80
Start Installation	.93
Full Operation Capability	1.0

Table 4 converts this to the appropriate descriptors and actual time schedule for this project.

Risk Analysis

The results of the SAVE simulation for development time, development effort and development cost can be shown in the form of probability plots. Assuming a normal (gaussian) distribution, all that is necessary is an estimate of the expected value (plotted at 50% level) and the standard deviation (plotted offset from the expected value at the 16 probability level) to generate the line. Then one can determine the probability of any value of the quantity in question. For ease of presentation, the plots are summarized in Table 5. For example, there is a 90% probability that the software development will not take more than 40 MY of effort. There is a 99% probability it will not take more than 45 MY of Development effort. There is only a 10 probability it will take less than 30 MY of Development effort.

The result for the development time is extremely important from a conceptual point of view. The small standard deviation is both a curse and a blessing. It says we can determine the development

Table 3.

SAVE SIMULATION

INPUT TO SIMULATION
$S_s = 98475, \quad \Sigma_{S_s} = 7081$
$VD = 15, \quad \Sigma_{VD} = 2$
Technology Constant = 10040
RESULTS OF SIMULATION
Number of iterations = 2170
Expected Development Time - 1.81 years
Standard Deviation, Development Time = .063 years
Expected Development Effort = 35.1 Manyears
Standard Deviation, Development Effort = 3.77 Manyears

Table 4.

SAVE MILESTONES

$t_d = 1.81$ years

EVENT	t/t_d	time from start (years)	time from start (months)
CDR	.43	.79	9
Software S.I.T.	.67	1.21	15
Hardware S.I.T.	.80	1.44	17
Start Installation	.93	1.68	20
Start Acceptance Test	1.0	1.81	22
Complete Acceptance Test	1.14	2.06	25

time very accurately ($\sigma_{t_d}/t_d \approx 3.5\%$) but at the same time it tells us we have little latitude in adjusting the development time to meet contractual requirements.

For example, σ_{t_d} = .063 years is .063 (52) = ± 3.28 weeks; $3\sigma_{t_d}$ = 3 (3.28) = ± 9.83 weeks;

= ± 9.83 ≈ ± 10 weeks

So, if we add 3σ to t_d we will be 99% sure that t_d will not exceed the actual value from random causes. This does not mean that requirements changes or late delivery of a computer will still permit the software to come in at ± 10 weeks of the expected time. These are external factors that will change t_d and must be specifically accounted for.

This is the curse. The system is very sensitive to external perturbations and these will generally cause development time increments greater than 2 or 3 σ_{t_d} (a 90 day delay in test bed computer delivery, say).

But, knowing this great time sensitivity, management can use it effectively in planning and contracting so that risk is always acceptable. The major point is: time is not a free good. Development time cannot be specified by management. The system determines that (i.e., software systems are inherently narrow band processes with sharp cutoff characteristics, a point that will be elaborated on later).

Manpower and Cash Flow Pattern

Now that we have the parameters for development effort and development time we can generate the manloading and cash flow pattern for the software development period (and even the life cycle, if we choose). The Rayleigh/Norden equation gives the instantaneous manpower.

$$\dot{y} = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2} \quad \text{MY/YR}$$

$$K = \hat{E}/.4 = 35.1/.4 = 87.75 \text{ MY}$$

$$t_d = 1.81 \text{ years}$$

so

$$\dot{y} = \frac{87.75}{(1.81)^2} \cdot t \cdot \exp\left(-t^2/2(1.81)^2\right) \quad \text{MY/YR}$$

for the software development effort (Phase II). The cash flow is just the average dollar cost/MY times $\dot{y}$.

$$\text{Cash Flow Phase II} = \overline{\$/\text{MY}} \cdot \dot{y} \quad \$/\text{YR}$$

Table 6 combines the software development effort (Phase II) with the initial design and system specification (Phase II) overlap and the hardware integration and test effort. The column labelled total adds the separate efforts together at each time period to show the total people on board. The cash flow rate is the annualized spending rate at that instant in time (assuming an average burdened cost/MY of $50,000). The last column gives the cumulative cost at each two month interval. Note that the total cumulative cost is greater than the predicted software development cost. This is because we have included the phase-out cost of Phase I effort plus the hardware planning integration effort.

Figure 3 shows the time-phased manloading of the Phase II part of the project as laid out in Table 6. Note that the overall curve is slightly distorted from the nominal software development effort because of Phase I phase-out and hardware planning efforts. Milestones are also shown on this figure to graphically portray where these should occur.

Linear Programming Alternative

An alternative method for the Rayleigh parameter determination is linear programming. Since we are dealing with only two unknowns, K and t_d, and have a number of constraint conditions involving these parameters, we can easily turn it into a two dimensional linear programming problem which can be solved graphically. The nice feature of this approach is that a number of the constraints can be expressed directly in management terms. Design to cost and design to contract time is possible within the constrained optimization procedure. This procedure is outlined below. The following constraint conditions apply:

$S_s = C_K K^{1/3} t_d^{4/3}$	Software equation
$K/t_d \le \sqrt{e}\ \dot{y}_{max}$	Maximum peak manpower
$K/t_d \ge \sqrt{e}\ \dot{y}_{max}$	Minimum peak manpower
$K/t_d^2 \le \lvert D \rvert$	Maximum difficulty
$K/t_d^3 \le \lvert \nabla D \rvert$	Maximum difficulty gradient

$t_d \le$ contract delivery time

$\overline{\$/\text{MY}}\ (.4K) \le$ Total budgeted amount for development

Table 5.

SUMMARY OF SAVE PROBABILITY PLOTS

Probability that value will not be greater than	DEV TIME (t_d) years	DEV EFFORT (E) Manyears	DEV COST (PH II) ($) Millions
1	1.55	25	1.25
10	1.73	30	1.50
20	1.76	32	1.60
50	1.81	35.1	1.76
80	1.86	38.5	1.93
90	1.90	40	2.04
99	1.97	45	2.25

Table 6.

EXPECTED SAVE MANLOADING & CASH FLOW RATE

t (Mos.)	PH II (People)	PH I (People)	Hdwre (People)	Total (People)	Cash Flow Rate	Cum. Cost ($ Million)
0	0	10	0	10	$ .5 M/yr	
2	5	8	0	13	$ .65 M/yr	.096
4	9	6	0	15	$ .80 M/yr	.217
6	13	4	1	18	.90	.358
8	17	0	2	19	.95	.513
10	21	0	2	23	1.15	.688
12	24	0	3	27	1.35	.896
14	26	0	3	29	1.45	1.129
16	28	0	4	32	1.60	1.383
18	29	0	4	33	1.65	1.654
20	30	0	4	34	1.70	1.933
22	30	0	6	36	1.80	2.225
24	20	0	4	24	1.20 M/yr	2.405

SAVE MANLOADING PLAN

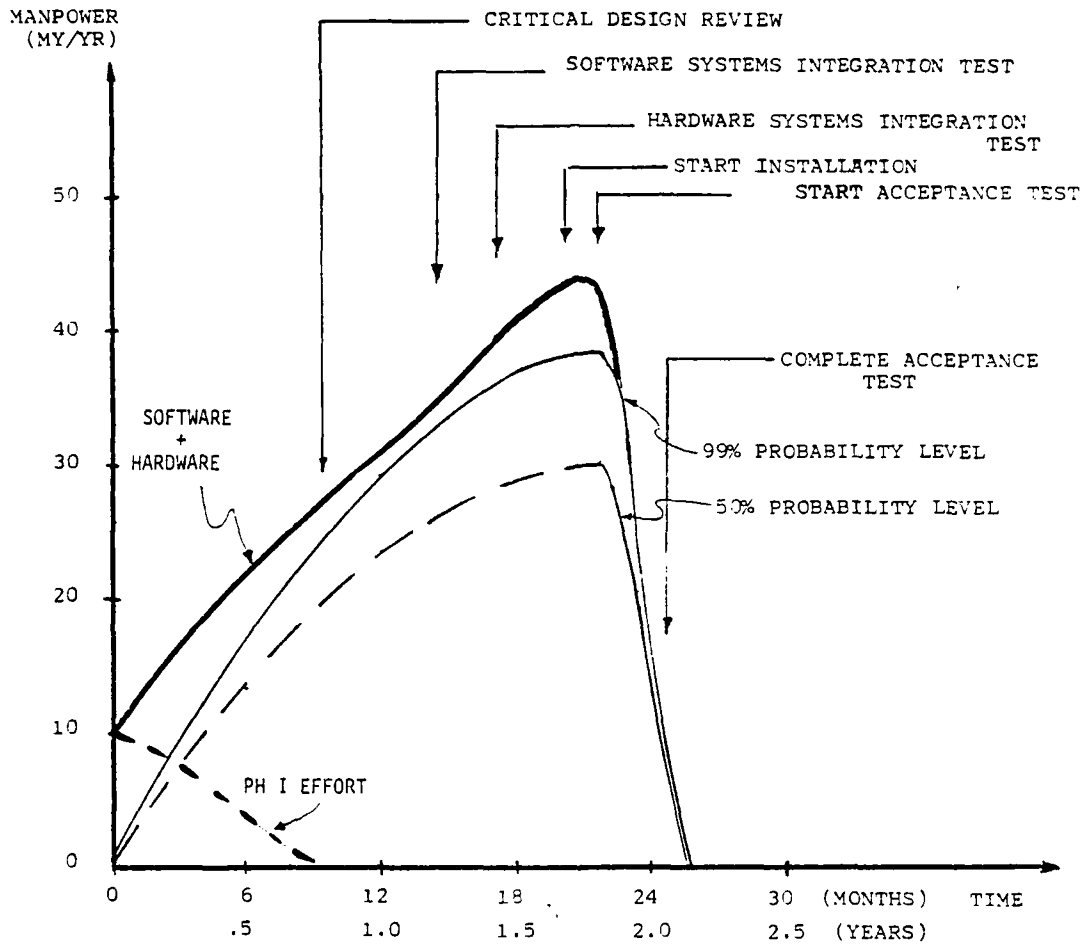


Figure 3

These constraint conditions can be linearized by taking logarithms and using the simplex method of solving the linear programming problem. The simplest objective functions are cost and time. One generally wants to minimize one or the other of these. Typically we do both and then trade-off in the region in between.

Assume these constraints applied to SAVE:

Number of S_s	$=$ 98475
Maximum development cost	$\leq$ \$2 million
Maximum time (contract delivery)	$\leq$ 2 years
Maximum manpower available at peak manning (hiring constraint, say)	$\leq$ 28 people
Minimum manpower you desire to employ at peak manning	$\geq$ 15 people
Maximum difficulty gradient	$\leq$ 15
Maximum difficulty	$\leq$ 50
Minimum productivity	$\geq$ 2000 S_s/ MY

These translate into:

$$1/3 \log K + 4/3 \log t_d = \log 98475 - \log 10040$$

$$\log K = \log (2 \times 10^6 / 5 \times 10^4 (.4))$$

$$\log t_d = \log 2$$

$$\log K - \log t_d = \log (\sqrt{e}\, 28)$$

$$\log K - \log t_d = \log (\sqrt{e}\, 15)$$

$$\log K - 3 \log t_d = \log 15$$

$$\log K - 2 \log t_d = \log 50$$

$$\log K = \log (98475/ .4(2000))$$

The intersection of these lines bound the feasible region. An optimal solution will be at some intersection point. Further, because of the equality constraint it must be along the S_s = 98475 line. The limiting conditions in this case are: $t_d \leq$ is 2 years, maximum peak manpower $\leq$ 28 people and S_s = 98475 source statements. Figure 4 shows the solution.

Reading off the solutions we see that:

	t_d (yrs)	K (MY)	E (MY)	$\overline{PR}$ (S_s/MY)
Minimum Time	1.83	84	33.6	98475/33.6 = 29[illegible]1
Minimum Cost	2.0	61	24.4 \$1.22M	98475/24.4 = 4036

Trade-off is possible along the S_s line between t_d = 1.83 years, K = 84 MY and t_d = 2 years, K = 61 MY without violating constraints.

Here it is easy to see the counterintuitive nature of productivity. Note that productivity increases with development time because the required effort (E) goes down as time is increased.

One other point is important. If the technology constant is smaller, the S_s = 98475 line would shift parallel to the right (direction of increasing time). If the constraints remained numerically the same, the feasible region would change because of the relocation of the S_s line. The time constraint could probably not be met and a relaxation of that constraint would have to be sought.

This is a deterministic solution. However, by extending the idea of simulation, the linear programming concept can be embedded within a simulation and the uncertain constraints can be allowed to vary randomly about their mean values and the statistical uncertainty for the minimum time and minimum cost solutions can be obtained by running the problem a few thousand times.

Results of the Example

We have shown that the management questions posed at the beginning can be answered quantitatively to acceptable engineering accuracy for a software project during the specification preparation phase. We need only know the state-of-technology we are going to apply to the development, estimate the number of lines of code using the PERT techniques, and use the software equation with a constraint relationship to solve for the management parameters (K, t_d) of the Rayleigh/Norden equation. Simulation provides suitable statistics for risk estimation.

IV. A Look at Fundamentals

Having shown some of the consequences of the Rayleigh equation and its application to the software process let's examine why we have selected this model rather than a number of other density functions that could be fitted to the data. We return to Norden's initial formulation to obtain the differential equation.

Norden's (8) description of the process is:

The rate of accomplishment is proportional to the pace of the work times the amount of work remaining to be done. This leads to the first order differential equation

LINEAR PROGRAMMING SOLUTION
FOR SAVE

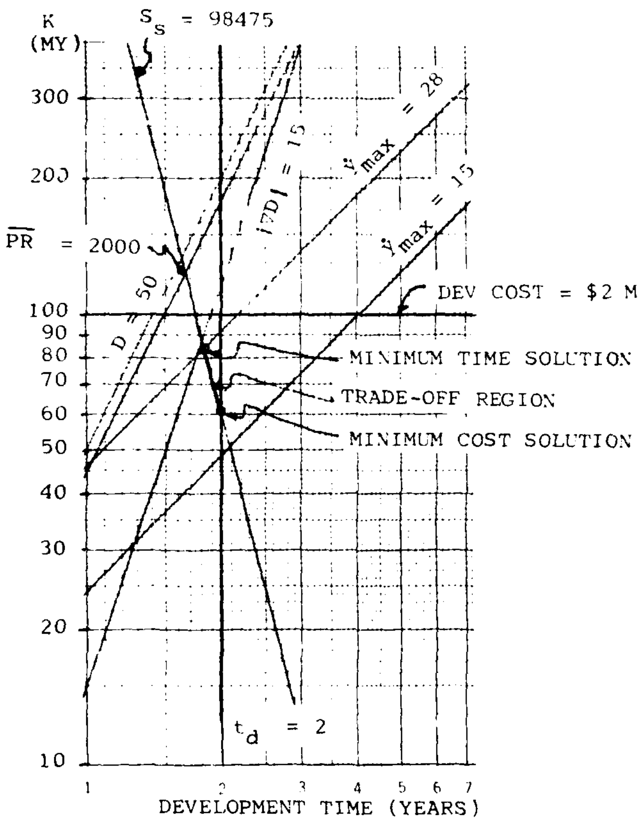


Figure 4

$\dot{y} = 2\ at\ (K-y)$

where $\dot{y}$ is the rate of accomplishment, 2at is the "pace" and (K-y) is the work remaining to be done. Making the explicit substitution for $a = 1/2\ t_d^2$ we have $\dot{y} = t/t_d^2\ (K-y)$. t/t_d^2 is Norden's linear learning law.

This is also the form of a generalized diffusion process related to Fick's equation of diffusion of mass transfer, Newton's equation of viscosity in momentum transfer and Fourier's equation of heat conduction (21,22).

The development by Mahajan and Schoeman (21) of the time pattern of the diffusion process is sketched below in the context of the spread of an innovation:

f(t) is the proportion of adopters at time, t

F(t) is the cumulative proportion of adopters at time t

$\bar{F}$ is the limit of the cumulative fraction of adopters

$$f(t) = \frac{d\ F(t)}{dt}$$

Now the rate of diffusion at any time is proportional to the potential adopters available at that time, or, in other words, as the cumulative number of adopters approaches its ceiling, $\bar{F}$, the rate of diffusion decreases proportionately. Thus

$$f(t) = \frac{d}{dt} F(t) \sim (F-F(t))$$

There is a "constant" of proportionality which will be labelled g(t) to indicate it may be a function having a time dependence. Then

$$f(t) = g(t)\ .\ (\bar{F}-F(t)).$$

This is the rate equation of the diffusion process. One initial condition is required to solve it. "The rate of diffusion...is controlled by g(t), the value of which depends on the specific innovation, the social system in which it is diffused and the channel and change agents used to diffuse it: g(t) = f (innovation, social system, channels, change agents). For a given innovation in a specific social system, the use of effective channels and change agents may be catalytic, affecting g(t) and, hence, the rate of diffusion --- g(t) may also be interpreted as the probability of an adoption at time t. Since $\bar{F}$-F(t) is the proportion...not adopted, the product g(t) . ($\bar{F}$-F(t)) gives the expected proportion of adopters at time t" (21).

We translate this into software terms and the Rayleigh parameters.

f(t) is $\dot{y}$ the rate of accomplishment, or manloading.

F(t) is y the cumulative fraction complete, or the cumulative effort.

$\bar{F}$ is K the ceiling, or the life cycle effort.

g(t) is Norden's "pace" (2at) and a function of the system (innovation), the software organization (social system), tools, constraints, state-of-technology being applied (channels), and the customer providing requirements, specifications and changes thereto (change agent).

Writing the diffusion equation in these Rayleigh/Norden terms we have

$$\dot{y} = g(t)\ \ (K-y)$$

$$\dot{y} = g(t)\ \ K-K(1-e^{-at^2})$$

$$\dot{y} = g(t)\ Ke^{-at^2}$$

If g(t) is equal to Norden's linear learning law then $g(t) = 2at = t/t_d^2$ where g(t) may be interpreted as the probability density for system development being completed at t_d. g(t) may also be thought of as the innovation frequency or the group problem solving frequency.

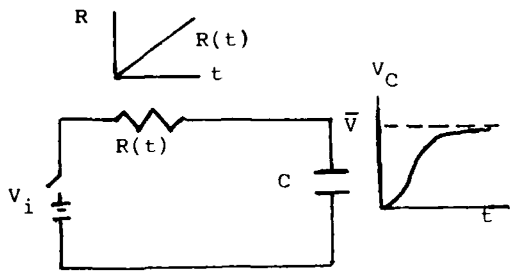
Substituting 2at for g(t) we have $\dot{y} = 2\ K\ ate^{-at^2}$, the derivative form of the Rayleigh equation.

Writing the equation in the diffusion form, we have $\dot{y} = t/t_d^2\ (K-y)$. We differentiate once more with respect to time and obtain

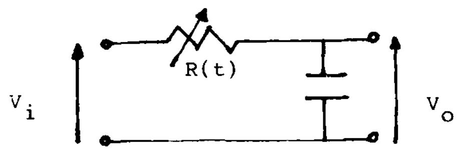
$$\ddot{y} = K/t_d^2 - y/t_d^2 + t(-\dot{y})/t_d^2\ , \text{ or}$$

$$\ddot{y} + t/t_d^2\ \dot{y} + y/t_d^2 = K/t_d^2 = D$$ which has also been obtained in a somewhat different development by Fix (23). This is a form of what we shall call the dynamic software equation. The Rayleigh integral is its solution which can be verified by direct substitution. Note that this equation is similar to the non-homogeneous 2d order differential equations frequently encountered in mechanical and electrical systems. There are two important differences. The forcing function, $D = K/t_d^2$, is a constant rather

than the more usual sinusoid and the $\dot{y}$ term has a variable coefficient, $t/_{t_d}2$, proportional to the 1st power of time. The differential equation immediately suggests the electrical and mechanical analogs. The electrical circuit is that for charging a capacitor with a constant voltage applied through a variable resistor whose resistance increases linearly with time.



We may also think of this as simple low pass filter.



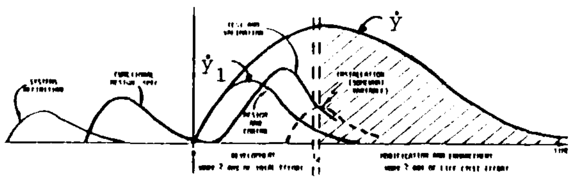
This comparison will become important later after we relate the differential equation more closely to the software process.

Systems Have a Bandwidth

It was stated earlier that the fundamental Rayleigh equation is the design and coding rate curve which is proportional to the rate of code production. The second order differential equation for this process expressed in manpower terms is

$$\ddot{y}_1 + \frac{6t}{t_d^2}\,\dot{y}_1 + \frac{6}{t_d^2}\,y_1 = \frac{K}{t_d^2}\,.$$

The sketch below shows the relation of the $\dot{y}_1$ curve to the overall manloading curve, $\dot{y}$.



The software differential equation

$$\ddot{y}_1 + 6t/_{t_d}2\;\dot{y}_1 + 6/_{t_d}2\;y_1 = K/_{t_d}2$$

is really a powerful narrow band filter. One can see this by perturbing the right hand side with a large magnitude but fast sinusoid. Very little ripple shows up on the $\dot{y}_1$ curve and y_1 is very smooth. A large random perturbation on the right hand side responds similarly, i.e., the system filters out all but the narrow band of frequencies the system is sensitive to.

This means that managers have to input work patterns that have frequencies that will be seen by the system, i.e., low frequency step functions vs. high frequency pulse functions (this may imply that short duration crash catch-up efforts are largely ineffective; in any event they have no long term cost and time increases).

Armed with these hints, one can take the Fourier transform of $\dot{y}_1$ curve and obtain the spectrum of frequencies the system will respond to. The Fourier transform of a Rayleigh equation in time is itself Rayleigh in frequency (radians/year). The transform is given in tables of Fourier Sine Transforms. Plotting the spectrum shows that it is indeed narrow and responds significantly only to signals having periods from 2 to 8 years, i.e., step-like functions, again for systems in the range of interest ($1 \le t_d \le 4$ years). In general, the pulse width of a management action has to be on the order of twice the development time to have long term consequences on the system.

This leads naturally to the idea that software systems have a characteristic system band width ($B_s \sim .6/t_d$). Then one can look at the problem in in terms of the signal to noise ratio and use the results given in texts on random signals and noise (17,18,19,20).

Uses of the Software Equation

The differential equation $\ddot{y} + t/t_d^2\,\dot{y} + 1/t_d^2\,y = K/t_d^2$ is very useful because it can be solved step-by-step using the Runge-Kutta solution. The solution can be perturbed at any point by changing $K/t_d^2 = D$, the difficulty. This is just what happens in the real world when the customer changes the requirements or specifications while development is in process. If we have an estimator for D (which we do) that relates K/t_d^2 to the system characteristics, say the number of files, number of reports and number of application programs, then we can calculate the change in D and add it to our original D, continue our Range-Kutta solution from that point in time and thus study the time slippage and cost growth consequences of such requirements changes. Several typical examples are given in reference 13.

When we convert this differential equation to the design and coding curve $\dot{y}_1$ by substituting $t_d / \sqrt{6}$, we obtain

$$\ddot{y}_1 + 6t/t_d^2 \; \dot{y}_1 + 6/t_d^2 \; y_1 = 6K_{1/t_d^2},$$

multiplying this by the $\overline{PR}$ and conversion factor as before, we obtain an expression that gives us the coding rate and cumulative code produced at time t.

$$\ddot{S}_s + \frac{t\,\dot{S}_s}{(t_{d/\sqrt{6}})^2} + \frac{S_s}{(t_{d/\sqrt{6}})^2} = 2.49 \;.\; \overline{PR} \;.\; K/t_d^2$$

$$= 2.49\; C_n\; (K/t_d^2)^{1/3}$$

$$\ddot{S} + \frac{6t}{t_d^2}\,\dot{S} + \frac{6}{t_d^2}\,S = 2.49\; C_n\; (D)^{1/3}$$

Since D is explicit, this equation can also be perturbed at any time t in the Runge-Kutta solution and changes in requirements studied relative to their impact on code production. Of course the earlier expressions for S_s and $\dot{S}_s$ could do the same.

An example of code production for SIDPERS using the Range-Kutta solution is given below.

V. Summary of Estimating Techniques at Various Stages of the Life Cycle

The following tables summarize the techniques that have been outlined and illustrated earlier in the paper. The adaptive filtering technique of Box (13,15) has not been repeated herein. It is important. Since it is adequately covered in the references, it is sufficient to say that the Rayleigh/Norden manpower curve is linearized by taking logarithms. Then the actual manpower (or coding rate) data can be fit by least squares, or plotted, to determine the parameters K and t_d. This technique has been applied to several hundred systems and works well, providing there is sufficient time history.

So, even though we have to use some sophisticated techniques to solve the problem conceptually, the actual implementation and obtaining of answers is mostly grocery store arithmetic with a little graphing (except for the Runge-Kutta solution which should be done by computer or programmable calculator; simulation should be done on a machine; Box method can be done graphically very nicely with good accuracy.)

We have shown that the management questions can be answered before software development starts and, perhaps more importantly, we can continually update and converge to the true behavior by adaptively using the real data stream in a dynamic way throughout the life cycle of the system. Thus, both prediction and control are possible to obtain and use in an engineering context.

Finally, we sum up with a few observations important to software managers.

<u>Software Lessons for Executives</u>

- SOFTWARE DEVELOPMENT IS DYNAMIC -- NOT STATIC
- PRODUCTIVITY RATES ARE CONTINUOUSLY VARYING -- NOT CONSTANT
- PRODUCTIVITY RATES ARE A FUNCTION OF THE SYSTEM DIFFICULTY -- MANAGEMENT CANNOT ARBITRARILY INCREASE PRODUCTIVITY
- BROOKS' LAW GOVERNS -- TIME AND MANPOWER ARE NOT FREELY INTERCHANGEABLE. (SHORTENING THE "NATURAL" DEVELOPMENT TIME OF A SYSTEM IS VERY COSTLY -- AND MAY BE IMPOSSIBLE)

Table 7.

Runge-Kutta Solution to the Coding Rate Differential Equation for SIDPERS

	t (years)	Coding Rate (S_s/year) (000)	Cumulative Code (S_s) (000)	
	0	0	0	
	.5	52.8	13.6	
	1.0	89.2	50.0	
	1.5	101.0	98.6	
	2.0	90.8	147.0	
	2.5	68.4	187.0	
	3.0	44.2	215.0	
	3.5	24.9	223.0	
$t_d \longrightarrow$	3.65	20.33	236.0	$\longleftarrow$ Actual size at extension was close to this
	4.0	12.3	241.0	
	4.5	5.36	246.0	
	5.0	2.09	247.0	

SIDPERS parameters

K	=	700 MY
t_d	=	3.65 years
D	=	52.54 MY/yr
PR	=	914 S_s/MY (burdened)

DIFFERENTIAL EQUATION

$$\ddot{S} + \frac{t}{\left(\frac{3.65}{\sqrt{6}}\right)^2}\dot{S} + \frac{1}{\left(\frac{3.65}{\sqrt{6}}\right)^2}S = 2.49.\ \overline{PR}\ .\ D = 2.49.(12009D^{-2/3}).D$$

$$= 2.49\ .\ (12009)\ .\ (D)^{1/3}$$

$$= 29902\ .\ (52.54)^{1/3}$$

$$= 111785$$

Table 8.

SUMMARY OF ESTIMATING EQUATIONS

Phase	Technique	Equation
Feasibility Functional Design	Rayleigh parameter estimation by: (a) Simulation (b) Linear Programming	(1) $S_s = C_k K^{1/3} t_d^{4/3}$ (2) $K/t_d^3 = \lvert \nabla D \rvert$ (3) $K/t_d^2 = \lvert D \rvert$ (4) $K/t_d = \sqrt{e}\, \dot{y}_{max}$ Solve (1) with (2), (3) or (4) (a) & (b) are conceptually the same.
Development	Initially, use result above: Box Technique: then fit actual overall manpower using Box technique. Fit code production rate using same technique. This will update parameters. Generate manloading curve.	$\ln(\dot{y}/t) = (K/t_d^2) - \frac{1}{2t_d^2} t^2$ Input $\dot{y}$, t, t^2; solve for t_d, K . Input $\dot{S}_s$, t, t^2; solve for $t_d/\sqrt{\epsilon}$, S_s
Operations & Maintenance	Box method as above: correlate with $K = E/_{.4}$, t_d actual, since these data are available from development.	$\overline{PR} = \frac{\text{Total } S_s}{E}$
Development, Operations & Maintenance	Requirements change. Use Range-Kutta method to access future impact.	$\ddot{y} + t/t_d^2\, \dot{y} + 1/t_d^2\, y = K/t_d^2 + \Delta D$ ΔD is usually positive; exception is when a portion of system is killed or taken away

Another table will help to specify the solution in terms of the management questions.

Table 9.

SUMMARY OF ESTIMATING TECHNIQUES FOR USE BY MANAGERS

Question	Answer
Can I do it?	Is calculated $t_d \leq$ contract time? (Yes - OK) Is calculated cost=$\$/\overline{MY}(.4K) \leq$ contract cost? (Yes - OK) Is difficulty and difficulty gradient reasonable for class of work? $\lvert D \rvert \leq 50$ unless there is a good precedent and lots of parallelism can occur. $\lvert \nabla D \rvert \leq$ 7 new 15 standalone 27 rebuild 55 composite Is C_k consistent with past experience, type system, language, tools, machine environment? (Yes - OK)
How much will it cost?	Dev. Cost = $\$/\overline{MY}(.4K)$
How long will it take?	Dev. time = $\hat{t}_d$ where $\hat{K}$, $\hat{t}_d$ come from $S_s = C_k K^{1/3} t_d^{4/3}$ and simultaneous solution of $K/t_d^2 = \lvert D \rvert$ $K/t_d^3 = \lvert \nabla D \rvert$ $K/t_d = \sqrt{e}\ \dot{y}_{max}$ Max cost = $\overline{\$}/MY(.4K)$ $t_d \leq$ contract time
How many people at any time?	$\dot{y}(t) = K/t_d^2 \cdot t \cdot e^{-t^2/2t_d^2}$ (MY/YR)
Cash flow?	$\$(t) = \overline{\$}/MY \cdot \dot{y}(t)$ (\$/YR)
What kind of skills?	Subjective by manager
What are the risks?	Standard deviations for K, t_d, $\dot{y}$, \$, E. E, t_d are plotted on probability paper to assess risk.

What are the trade-offs?	$E = \dfrac{\left(\frac{S_s}{C_k}\right)^3 .4}{t_d^4}$ $\$ \text{ DEV} = \dfrac{\overline{\$/MY} \left(\frac{S_s}{C_k}\right)^3 .4}{t_d^4}$
How do constraints affect the answers?	If constraints cannot be met, trade-off law has to be invoked to get K, t_d in a reasonable range. Lengthening t_d will usually do this. If can't meet time, gradient constraint, project is not do-able as is. Size of system (S_s) has to be reduced.
What is the cost and schedule impact of a requirements change (during development)?	At the time of impact, solve $\ddot{y} + t/t_d^2\ \dot{y} + 1/t_d^2\ y = D + \Delta D$ for $\dot{y}$, y. Plot for all t, compare with previous $\dot{y}$, y curves. $y_{new}(\infty) = K_{new}$ $y_{old}(\infty) = K_{old}$ $K_{new} - K_{old} = \Delta K$ $\Delta E = \Delta K\ (.4)$. Δ LC Cost = $\overline{\$/MY} \cdot \Delta K$; Δ DEV cost = $\overline{\$/MY} \cdot \Delta E$ $\dot{y}_{new}$ (peak) gives new t_d. (Lagrangian 3 point interpolation can give this precisely.) then Δt_d = slippage $= t_{d_{new}} - t_{d_{old}}$.

Glossary of Notation

K	- Rayleigh/Nordon life cycle effort parameter; work units, man years, man months, etc.
t_d	- Rayleigh/Norden time parameter. Time which peak manpower normally occurs for large software projects. Mathematically the peak of the curve, $\dot{y} = K/t_d^2 \,.\, t \,.\, e^{-t^2/2t_d^2}$.
E	- Development effort. Area under Rayleigh manpower curve up to t_d (peak). Nominally, $E = .4K$.
$\dot{y}_{max}$	- Peak manpower. Normally occurs at t_d. $\dot{y}_{max} = \frac{K}{\sqrt{e}\, t_d}$
K/t_d^2	- System Difficulty, a force-like term. Ratio of effort to development time squared.
K/t_d^3	- Proportional to magnitude of difficulty gradient. $\nabla D = \hat{i}\,(-2K/t_d 3) + \hat{j}(1/t_d 2)$ $\lvert \nabla D \rvert = \sqrt{\left(\frac{-2K}{t_d^3}\right)^2 + \left(1/t_d^2\right)^2}$ $\approx 2K/t_d^3$ since $1/t_d^2$ is always a small number in the systems context.
$\overline{PR}$	- Productivity. Defined as the total number of source statements (S_s) divided by the development effort (E). $\overline{PR} = \frac{S_s}{E} = \frac{S_s}{.4K}$ source statements/MY. This is generally a burdened number which includes overhead (non-programming) effort.
S_s	- No. of delivered lines of executable source code. Does not include comments. What the programmer writes as opposed to machine language instructions.
C_k	- A state of technology constant. Relates to the software equation $S_s = C_k K^{1/3} t_d^{4/3}$. A channel capacity constraint in the information theory sense. (May be thought of as the diameter of the "pipe" regulating the "flow" of source code).
$\overline{\$/MY}$	- Average dollar cost per man year of effort.
$ DEV	- Development cost in dollars. $\$\ DEV = \overline{\$/MY} \,.\, E$.
y	- Cumulative effort at time t. (man years).
$\dot{y}$	- Manpower at time t. (MY/YR).
$\ddot{y}$	- Rate of change of manpower at time t. A force-like term. (man years per year^2).

References

1. Brooks, F.P. Jr., The Mythical Man-Month, Addison-Wesley Publishing Co., Reading, MA. 1975.

2. Morin, Lois H., Estimation of Resources for Computer Programming Projects, MS Thesis, Univ. of North Carolina, Chapel Hill, N.C., 1973.

3. Gehring, Phillip F., and Pooch, Udo W., "Software Development Management," Data Management, Feb. 1977, pp. 14-18.

4. Gehring, Phillip F., Improving Software Development Estimates of Time and Cost, Paper presented to 2nd International Conference on Software Engineering, San Francisco, 13 Oct. 1976.

5. Gehring, P.F., A Quantitative Analysis of Estimating Accuracy in Software Development, Ph.D. Thesis, Texas Univ., College Station, TX, Aug. 1976.

6. Aron, Joel D., A Subjective Evaluation of Selected Program Development Tools, Papers of the Software Life Cycle Management Workshop, Airlie, Va., Aug. 1977, sponsored by US Army Computer Systems Command.

7. Norden, Peter V., "Useful Tools for Project Management." Management of Production, M.K. Starr (Editor), Penguin Books, Inc., Baltimore, Md., 1970, pp. 71-101.

8. Norden, Peter V., Project Life Cycle Modelling: Background and Application Of The Life Cycle Curves, Papers from the Software Life Cycle Management Workshop, Airlie, Va., Aug. 1977, sponsored by US Army Computer Systems Command.

9. Putnam, Lawrence H., "A Macro-Estimating Methodology for Software Development," Digest of Papers, Fall COMPCON '76, Thirteenth IEEE Computer Society International Conference, Sept. 1976, pp. 138-143.

10. Putnam, Lawrence H., "ADP Resource Estimating: A Macro-Level Forecasting Methodology for Software Development," Proceedings of the Fifteenth Annual US Army Operations Research Symposium, 26-29 Oct. 1976, Forher, Va., pp. 323-327.

11. Putnam, Lawrence H., A General Solution to the Software Sizing and Estimating Problem, as presented at the Life Cycle Management Conference, American Institute of Industrial Engineers, Wash., D.C., 8 Feb. 1977.

12. Putnam, Lawrence H., "The Influence of the Time-Difficulty Factor in Large Scale Software Development," Digest of Papers, IEEE Fall COMPCON '77 Fifteenth IEEE Computer Society International Conference, Sept. 1977, Wash., D.C. pp. 348-353.

13. Putnam, Lawrence H., and Wolverton, Ray W., Quantitative Management: Software Costing Estimating, *A Tutorial for COMPSAC '77*, The IEEE Computer Society's First International Computer Software and Applications Conference, Chicago, Ill., 8-10 Nov. 1977.

14. Putnam, Lawrence H., "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," to appear in IEEE Transactions on Software Engineering, Summer, 1978.

15. Box, George E.P and Pallesen, Lars, Software Budgeting Model, Mathematics Research Center, University of Wisconsin, Madison (Feb. 1977 pre-publication draft).

16. DA Pamphlet No. 18-8, Management Information System, A Software Resource Macro-estimating Procedure, Hq. Dept. of the Army, Feb. 1977.

17. Carlson, A. Bruce, Communication Systems: An Introduction to Signals and Noise in Electrical Communication, McGraw-Hill, Inc., New York, 1968.

18. Schwartz, Mischa, Information Transmission, Modulation, and Noise, McGraw-Hill Book Company, second edition, New York, 1970.

19. Thomas, John B., An Introduction to Statistical Communication Theory, John Wiley & Sons, Inc., New York.

20. Baghdady, Elie J., editor, Lectures on Communication Systems Theory, McGraw-Hill Book Company, Inc., New York, 1961.

21. Mahajan, Vijay and Schoeman, Milton E.F., Generalized Model for the Time Pattern of the Diffusion Process, IEEE Transactions on Engineering Management, Vol. EM-24, No. 1, February 1977, pp. 12-18.

22. Furth, R.H., Fundamentals Principles of Modern Theoretical Physics, Permagon Press, Oxford, 1966.

23. Fix, George J., The Dynamics of Software Development I, Papers from the Software Life Cycle Management Workshop, Airlie, Va., Aug. 1977, sponsored by the US Army Computer Systems Command.

SOFTWARE COST MODELING: SOME LESSONS LEARNED

B.W. Boehm and R.W. Wolverton
TRW Defense and Space Systems Group
Redondo Beach, CA 90278
August 1978

ABSTRACT

This paper sumarizes some of the lessons we learned recently in developing a software cost estimation model for TRW. With respect to the AIRMICS Workshop, we will concentrate on three issues which we found particularly important and useful to address in an industry-wide context at the Workshop. These issues are:

1. There is a need to develop a set of well-defined, agreed-on criteria for the "goodness" of a software cost model.
2. There is a need to evaluate existing and future models with respect to these criteria.
3. There is a need to emphasize "constructive" models which relate their cost estimates to actual software phenomenology and project dynamics.

1. CRITERIA FOR THE GOODNESS OF A SOFTWARE COST MODEL

Our initial criteria reflected our primary objective in developing the new TRW Software Cost Estimating Program (SCEP): *to serve as an aid in developing cost estimates for* competitive proposals on large Government software projects. One may have other objectives for developing a cost model e.g. to evaluate the impact of using new techniques which may require a somewhat different set of criteria.

Here is a list of the criteria we found important in our context:

1. Definition. Has the model clearly defined which costs it is estimating, and which costs it is excluding?
2. Fidelity. Are the estimates close to the actual costs expended on the projects?
3. Objectivity. Does the model avoid allocating most of the software cost variance to poorly-calibrated subjective factors (e.g. complexity)? That is, is it hard to juggle the model to get any result you want?
4. Constructiveness. Can a user tell why the model gives the estimates it does? Does it help him understand the software job to be done?
5. Detail. Does the model easily accommodate the estimation of a software system consisting of a number of subsystems and units? Does it give (accurate) phase and activity breakdowns?
6. Stability. Do small differences in inputs produce small differences in output cost estimates?
7. Scope. Does the model cover the class of software projects whose costs you need to estimate?
8. Ease of Use. Are the model inputs and options easy to understand and specify?
9. Prospectiveness. Does the model avoid the use of information which will not be well known until the project is complete?*

In the process of developing the TRW SCEP model and evaluating existing models (Refs. 1-10), we found each of these criteria important in terms of lessons we were learning about software cost modeling. Below are some of the results of our evaluation and model development with respect to the first four criteria.

1.1 Definition.

Where we ran a software cost model to support a cost proposal one of the first questions we would get from the proposal manager was, "Does this estimate include the cost of management? requirements analysis? training? computer operators?" etc. We were somewhat surprised to find that the documentation for most existing cost models doesn't satisfactorily answer this question. For the TRW SCEP model, we found that the best solution was to define a standard-form Work Breakdown Structure (Fig. 1) *and use it to define which costs were included in our estimates* and which were excluded.

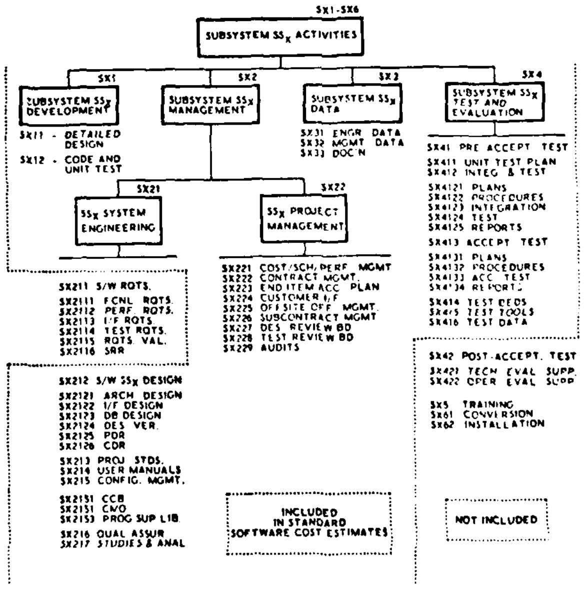


Fig. 1. Work Breakdown Structure Activity Hierarchy

*Clearly, this criterion is specific to the objective of cost prediction. For other objectives, such as technology impact assessment, a retrospective model could be appropriate.

1.2 Fidelity.

We found that some existing models seemed to work well for some classes of software and not very well for others. For example, Fig. 2 presents the results of our analysis of the Boeing[9] and Putnam[10] estimation models when applied to their own (Boeing and US Army Computer Systems Command) data. It is seen that the Boeing model appears to estimate small projects reasonably well, but gives extreme overestimates on large projects, while the Putnam model does just the reverse. It would be highly valuable for the field if similar information were available with respect to other models and other software project attributes.

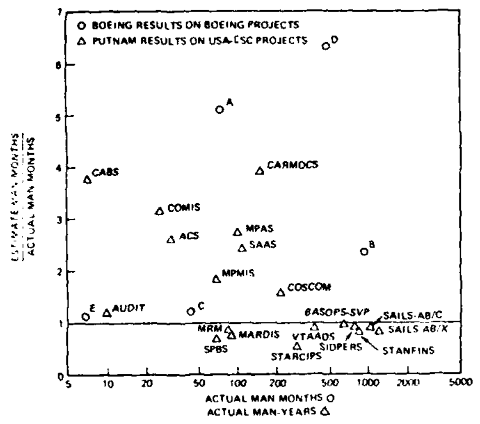


Fig. 2. Estimator Fidelity versus Project Size: Boeing and Putnam Models

For comparison, Fig. 3 shows the performance of the TRW SCEP model on 20 completed TRW projects.

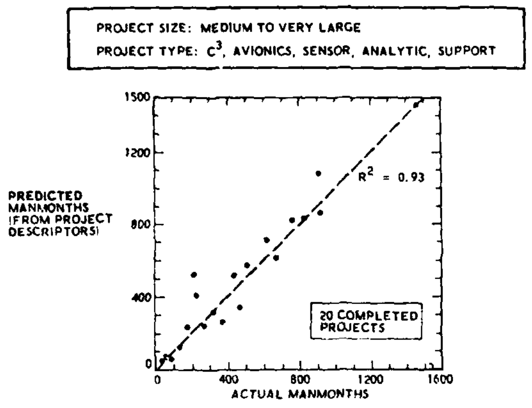


Fig. 3. Example of TRW Software Cost Model Performance

The sequence of activities in developing, calibrating and evaluating the model was as follows:

a. Survey the current state of the art in cost estimation. Fig. 4 shows one of the results of the survey, a summary of the factors used in the major current cost estimation models.

b. Develop a baseline model. The phase-sensitive nature of the model is based on a model developed earlier by the authors to evaluate the cost impact of new software technologies;[11] it is functionally similar to the Boeing model[9] in this regard.

c. Refine the model parameters via a two-round Delphi exercise involving 10 experienced TRW line and project managers.

d. Calibrate the model parameters using an initial sample of seven completed projects.

e. Evaluate the model using a sample of 20 completed projects, including the initial seven. This evaluation was done by an independent third party.

GROUP	FACTOR	SDC 65	TRW 72	PUTNAM	DOTY	RCA	IBM	BOEING	TRW 78
SIZE	SOURCE INST.			X	X		X	X	
	OBJECT INST.	X	X		X	X			X
	NO. ROUTINES	X		X		X			
	NO. DATA ITEMS			X			X		X
	NO. REPORTS			X					
	DOCUMENTATION				X		X		
	NO. PERSONNEL			X			X	X	
PROGRAM ATTRIB.	TYPE	X	X	X	X	X	X	X	X
	COMPLEXITY		X			X	X		X
	LANGUAGE	X						X	X
	RE-USE			X		X		X	X
HARDWARE ATTRIB.	TIME CONSTRAINT		X	X	X	X	X	X	X
	STORAGE CONSTRAINT				X	X	X		X
	H/W CONFIGURATION	X				X			
	CONCURRENT H/W DEV.	X			X		X		X
PROJECT ATTRIB.	PERSONNEL QUALITY						X		X
	PERS. CONTINUITY			X			X		
	H/W EXPERIENCE	X			X		X		X
	APPLIC. EXPER.		X	X			X	X	X
	LANGUAGE EXPER.						X		
	TOOLS & TECHNIQUES			X		X	X	X	
ENVIR. ATTRIB.	CUSTOMER I/F	X					X		
	RQTS. DEFINITION	X			X		X		
	RQTS. VOLATILITY	X			X		X		X
	SCHEDULE			X		X			
	SECURITY						X		X
	COMPUTER ACCESS			X	X		X	X	X
	TRAVEL/REHOSTING	X		X	X				
	REQUIRED QUALITY					X			X

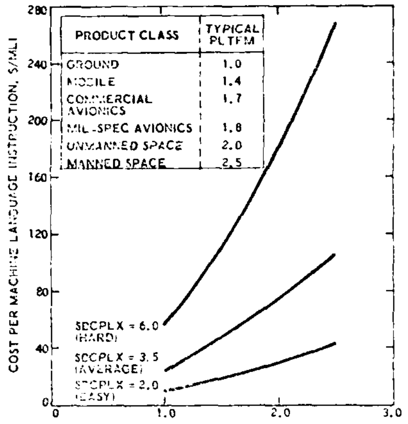
Fig. 4. Factors Used in Various Cost Models

The high value of R^2, the square of the correlation coefficient, should be interpreted with considerable care. For example, the sample consists of only medium-to-very large TRW government contract software efforts; we are not sure how it performs on other types of software effort. Further, the calibration is on past projects. We have included a factor to cover the impact of software technology improvements; however, until we get more experience in comparing SCEP estimates with the resulting project actuals, we must consider the model still in an experimental state.

1.3 Objectivity.

Figure 5 shows one of the results of our analysis of the RCA PRICE S model in September 1977. (It is possible that the model may have been subsequently changed). It shows the extreme sensitivity of the model to the subjective complexity (SDCPLX) factor. If you describe a project as "HARD", the model will produce a cost estimate that is 6-7 times higher than if you describe the project as "EASY". This is a huge source of variation for a parameter that is entirely subjective. As we have found from experience, it means that a user can make the model produce any cost estimate he wants, simply by modifying the subjective complexity factor. (In fact, PRICE S has a mode called ECIRP which will do this for you automatically). Doing this may solve a user's short-term pricing problem, but it hardly leaves you with the feeling that you have performed any objective and meaningful cost estimation function for the user.

In developing the TRW SCEP model, we found that we were unable to avoid including a complexity factor. However, we have made the complexity rating an attribute of each individual unit in the software, and provide users with a set of scales for calibrating the complexity of different types of units. Figure 6 shows the complexity scale for computational/numerical analysis type units as an example. Having such a scale makes the complexity rating a much more objective, verifiable attribute.



PRODUCT CLASS	TYPICAL PLTFM
GROUND	1.0
MOBILE	1.4
COMMERCIAL AVIONICS	1.7
MIL-SPEC AVIONICS	1.8
UNMANNED SPACE	2.0
MANNED SPACE	2.5

Fig. 5. Effect of Complexity Factor in RCA Price S Model (Sept. 1977)

1.4 Constructiveness

We use the TRW SCEP model as a means of cross-checking separate cost estimates developed by project personnel *in different* ways. Inevitably, this leads to differences between the various estimates, and a need to understand why one was higher or lower than the other. Similarly, project personnel want to know in project terms why different factor ratings give them different cost estimates.

In order to help answer these questions, and in order to promote accurate factor ratings by users, we have provided a table for each of the factors in the TRW SCEP model that shows the impact on project activities of the various factors and their ratings. For example, the table for the "Required Quality" Factor is shown as Figure 7. When a user rates his system as having a "Very High" Required Quality, the SCEP model will increase the cost of performing the various phases of system development by a certain factor, which may vary by phase. The *table in Figure 7* thus tells the user why his costs have increased, in terms of the impact of a very high Required Quality on his project activities. In general, we have found that such tables do help the users of the cost model better understand the software job they are preparing for. Similarly, we feel they provide a link between *the "Predictive Models" and "Life Cycle Dynamics"* portions of the Workshop which deserve discussion and explanation during the Workshop.

		EXAMPLE RATINGS FOR COMPUTATIONAL-TYPE UNIT
1	• VERY LOW	• COMPUTATIONS TO EVALUATE SIMPLE EXPRESSIONS: A = B + C * (D - E)
2	• LOW	• USE OF STANDARD MATH AND STATISTICAL ROUTINES AND BASIC MATRIX-VECTOR OPERATIONS
3	• AVERAGE	• USE OF BASIC NUMERICAL ANALYSIS SKILLS E.G., MULTIVARIATE INTERPOLATION ORDINARY DIFFERENTIAL EQUATIONS
4	• HIGH	• DIFFICULT BUT STRUCTURED NUMERICAL ANALYSIS E.G., NEAR-SINGULAR MATRIX EQUATIONS, PARTIAL DIFFERENTIAL EQUATIONS
5	• VERY HIGH	• DIFFICULT, UNSTRUCTURED NUMERICAL ANALYSIS E.G., FAST, HIGHLY ACCURATE ANALYSIS OF NOISY, STOCHASTIC DATA

Fig. 6. Example of Complexity Scale: TRW Cost Model

1.5 Other Factors.

We have encountered similar lessons learned with respect to other factors in the above list: detail, stability, etc. The length constraints of this paper preclude our discussing them in detail here; however, we hope they can serve as topics for discussion at the Workshop.

2. SUGGESTED ISSUES FOR WORKSHOP

Based on the discussion above, here are some suggested questions which would be useful to discuss at the Workshop.

a. What criteria are important for the utility of a software cost estimation model?

b. In general, how do current predictive models stack up with respect to these criteria? Are there general deficiencies in current models (e.g., uniform or comparable definitions) which need attention?

FACTOR	RATING	RQTS. ANALYSES	PRELIM. DESIGN	DETAIL DESIGN	CODE & UNIT TEST	INTEG. & TEST
1. REQUIRED QUALITY	VL	•LITTLE DETAIL •MANY TBD'S •LITTLE VALID'N •MINIMAL PLANS (QA, CM, ACCEPT.) •MINIMAL SRR	•LITTLE DETAIL •MANY TBD'S •LITTLE VERIF'N •MINIMAL QA, CM, STDS. •MINIMAL PDR •NO DRAFT USER MAN.	•BASIC DESIGN INFO. •MINIMAL UDF'S •MINIMAL QA, CM •MINIMAL CDR •NO DRAFT USER MAN.	•MINIMAL U.T. PLAN •MINIMAL FCL'S, PATH TEST, STDS CHECK •MINIMAL QA, CM •MINIMAL I/O AND OFF-NOMINAL TESTS •MINIMAL USER MAN.	•MINIMAL TEST PLANS •NO TEST PROCEDURES •MANY RQTS. UNTESTED •MINIMAL QA, CM •MINIMAL STRESS, OFF-NOMINAL TESTS •MINIMAL AS BUILT DOCU.
	L	•BASIC INFO. VALID'N •FREQUENT TBD'S •BASIC PLANS (QA, CM, ACCEPT.) •BASIC SRR	•BASIC INFO. VERIF'N •FREQUENT TBD'S •BASIC QA, CM, STDS •BASIC PDR •MINIMAL USER MAN.	•MODERATE DETAIL •BASIC UDF'S, PA, CM, CDR •MINIMAL USER MAN.	•BASIC U.T. PLAN •PARTIAL FCL'S, PATH TEST, STDS CHECK •BASIC QA, CM, U.M. •PARTIAL I/O AND OFF-NOMINAL TESTS	•BASIC TEST PLANS •MINIMAL TEST PROCED. •FREQUENT RQTS UNTESTED •BASIC QA, CM, U.M. •PARTIAL STRESS, OFF-NOMINAL TESTS
	S	FULL TRW POLICIES	FULL TRW POLICIES	FULL TRW POLICIES	FULL TRW POLICIES	FULL TRW POLICIES
	H	•DETAILED VALID'N, PLANS, SRR	•DETAILED VERIF'N, QA, CM, STDS, PDR, DOC'N	•DETAILED VERIF'N, QA, CM, STDS, CDR DOC'N	•DETAILED U.T. PLAN, FCL'S, QA, CM, DOC'N CODE WALKTHRUS •EXTENSIVE OFF-NOMINAL TESTS	•DETAILED TEST PLANS, PROCEDURES, QA, CM, DOC'N •EXTENSIVE STRESS, OFF-NOMINAL TESTS
	VH	•DETAILED VALID'N, PLANS, SRR •IV&V INTERFACE (SUPPORT, RESPONSE)	•DETAILED VERIF'N, QA, CM, STDS, PDR, DOC'N •IV&V INTERFACE	•DETAILED VERIF'N, QA, CM, STDS, CDR, DOC'N •IV&V INTERFACE	•DETAILED U.T. PLAN, (FCL'S, QA, CM, DOC'N) CODE WALKTHRUS •VERY EXTENSIVE OFF-NOMINAL TESTS •IV&V INTERFACE	•VERY DETAILED TEST PLANS, PROC'S, QA, CM, DOC'N •VERY EXTENSIVE STRESS, OFF-NOMINAL TESTS •IV&V INTERFACE

Fig. 7. Example of Factor/Rating Definitions: TRW Cost Model

c. To what extent do current predictive models explain software life-cycle dynamics? Are there ways to help close the gap between models and project dynamics?

REFERENCES

1. E.A. Nelson, Management Handbook for the Estimation of Computer Programming Costs, SDC, AD-A648750, 31 October 1966.

2. R.W. Wolverton, "The Cost of Developing Large-scale Software," IEEE Trans. on Computers, pp. 615-636, June 1974.

3. L.H. Putnam, A General Solution to the Software Sizing and Estimating Problem, as presented at the Life Cycle Management Conference, AIIE, Washington, D.C. February 8, 1977.

4. J.R. Herd, J.N. Postak, W.E. Russell, and K.R. Stewart, Software Cost Estimation Study – Study Results, Doty Associates, Inc., Final Technical Report, RADC-TR-77-220, Vol. I (of two), June 1977; NTIS No. AD-A042264.

5. D.L. Doty, P.J. Nelson, and K.R. Stewart, Software Cost Estimation Study – Guidelines for Improved Software Cost Estimating, Doty Associates, Inc., Final Technical Report, RADC-TR-77-220, Vol. II (of two), August 1977; NTIS No. AD-A044609; see also errata sheet, J.H. Herd Doty Associates, Inc., 416 Hungerford Dr., Rockville, MD 20850, 6 June 1978.

6. John Schneider, IV, A Preliminary Calibration of the RCA PRICE/S Software Cost Estimation Model, NTIS No. AD-A046808, September 1977.

7. J.D. Aron, Estimating Resources for Large Programming Systems, NATO Science Committee, Rome, Italy, October 1969.

8. C.P. Felix and C.E. Walston, "A Method of Programming Measurement and Estimation," IBM Sys. J., Vol. 16, No. 1, 1977.

9. R.K.D. Black, R.P. Curnow, R. Katz, and M.D. Gray, BCS Software Production Data, Boeing Computer Services, Inc., Final Technical Report, RADC-TR-77-116, March 1977; NTIS No. AD-A039852.

10. L.H. Putnam and R.W. Wolverton, "Quantitative Management: Software Cost Estimating," IEEE Comp. Soc. First Int'l Computer Software and Applications Conf. (COMPSAC 77), IEEE Catalog No. EHO 129-7, Chicago, IL, Nov. 8-11, 1977, 326 pp.

11. B.W. Boehm, C.A. Bosch, A.S. Liddle, and R.W. Wolverton, "The Impact of New Technologies on Software Configuration Management," TRW Report to USAF-ESD, Contract F19628-74-C-0154, 10 June 1974.

A SOFTWARE ERROR DETECTION MODEL WITH APPLICATIONS

Amrit L. Goel
Syracuse University
Syracuse, New York, USA

Abstract

This paper deals with the modelling of software errors encountered in a small and a large software system. A deterministic analysis of software failure process is presented to obtain an appropriate mean value function for a non-homogeneous Poisson process. Several quantitative measures for software quality assessment are also proposed. Statistical techniques of inference about unknown parameters are discussed and detailed analyses of software error data from two systems are presented.

1. Introduction

The importance of modelling and analysis of software error phenomena has been well recognized during the last few years and many studies have addressed this problem, see for example [1,4,5,6,7,8,9, 10,12,13,14,15,16,17]. An important objective of most of these investigations has been to develop analytical models for the error phenomenon in order to compute quantities of interest such as the number of errors detected by some time t, the number of remaining errors at time t, the software reliability function, and the mean time to software failure. Such quantities are useful for planning purposes, both in the development and the operational phases of software systems.

In this paper we develop a time dependent model for software errors and illustrate its applicability via analyses of error data from two software projects. The model considered here is a non-homogenious Poisson process whose mean value function is derived by a deterministic analysis of the software failure process in Section 2. Several quantities of interest are then given to establish quantitative measures for software performance. A description of the first project (a large scale software project) and error analyses are given in Section 3. Software error data based on CPU time from a small project is given in Section 4.

2. Time Dependent Model and Quantities of Interest

2.1 Model Development

Before analyzing the stochastic behavior of the software failures, it is useful to make a simpler analysis ignoring statistical fluctuations in the number of software failures. Suppose $n(t)$ is the cumulative number of software errors detected by time t and $n(t)$ is so large that it can be treated as a continuous function of t. We assume that the number of undetected errors at any time is finite and hence $n(t)$ is a bounded, non-decreasing function of t. We further assume that the total number of errors to be eventually detected, $n(\infty)=a$. Then, we have

$$n(t) = \begin{cases} 0 & \text{when } t=0 \\ a & \text{when } t=\infty \end{cases} . \qquad (1)$$

Now let the number of errors detected in $(t,t+\Delta t)$ be proportional to the number of undetected errors, i.e.

$$n(t+\Delta t)-n(t) = b\{a-n(t)\}\Delta t \qquad (2)$$

where b is a constant of proportionality. From (2) we get the differential equation

$$n'(t) = ab-bn(t). \qquad (3)$$

Solving this for $n(t)$, we get

$$n(t) = a(1-e^{-bt}). \qquad (4)$$

A stochastic analysis of this phenomenon using a time dependent Poisson process (a non-homogeneous Poisson process) and the role of $n(t)$ in such analysis is presented next.

Let $\{N(t),t\geq 0\}$ be a counting process (number of errors in $(0,t]$). The difference between $n(t)$ and $N(t)$ is that the former is a deterministic number while the latter is a random variable. The $N(t)$ process is a NHPP with intensity function $\lambda(t)$ if

(i) $N(0) = 0$
(ii) $\{N(t),t\geq 0\}$ has independent increments
(iii) $P\{2 \text{ or more events in } (t,t+h)\}=o(h)$
(iv) $P\{\text{exactly 1 event in } (t,t+h)\} = \lambda(t)h+o(h)$.

If we let

$$m(t) = \int_0^t \lambda(s)ds \qquad (5)$$

then it can be shown that

$$P\{N(t)=n\} = \frac{\{m(t)\}^n}{n!} e^{-m(t)} \qquad n \geq 0. \qquad (6)$$

In other words, $N(t)$ has a Poisson distribution with expected value $E[N(t)]=m(t)$ for $t>0$ and $m(t)$ is called the mean value function of the NHPP.

The deterministic model (4) derived above has been found to be a good descriptor of the software failure process when applied to actual data sets. For this reason we choose the mean value function to be

$$m(t) = a(1-e^{-bt}). \qquad (7)$$

Note that

$$EN(\infty) = m(\infty) = a. \qquad (8)$$

In other words, the parameter 'a' can be interpreted as the expected total number of software errors to be detected until complete debugging, assuming such debugging to be perfect whenever an error occurs.

2.2 Quantities of Interest

We obtain expressions for the following quantities to establish the performance measures for software reliability assessment. For

Cumulative number of software errors. For given a and b the distribution of $N(t)$, the cumulative number of software errors detected by time t, is given by

$$P\{N(t)=n\} = \frac{\{a(1-e^{-bt})\}^n}{n!} e^{-a(1-e^{-bt})}$$

$$n=0,1,2,\ldots \qquad (9)$$

i.e. $N(t)$ has a Poisson distribution with mean

$$EN(t) = a(1-e^{-bt}). \qquad (10)$$

Note that

$$P\{N(\infty)=n\} = \frac{a^n}{n!} e^{-a}, \quad n=0,1,2,\ldots \qquad (11)$$

i.e. the distribution of $N(\infty)$, the total number of errors to be detected if debugging is carried out indefinitely, is also a Poisson distribution with mean 'a'.

Remaining number of software errors and related results. Let $\bar{N}(t)$ be the number of errors remaining in the system at time t. Then

$$\bar{N}(t) = N(\infty)-N(t), \qquad (12)$$

$$E\bar{N}(t) = ae^{-bt}, \qquad (13)$$

and

$$\mathrm{Var}(\bar{N}(t))=\mathrm{Var}(N(\infty))+\mathrm{Var}(N(t))-$$
$$- 2\mathrm{Cov}(N(t),N(\infty))$$

$$= a+a(1-e^{-bt})-2a(1-e^{-bt})$$

or $\mathrm{Var}(\bar{N}(t)) = ae^{-bt}$. (14)

Now suppose y_n is the number of errors found in a testing period t_n, i.e. $N(t_n)=y_n$. Then the conditional distribution of $\bar{N}(t_n)$ is

$$P\{\bar{N}(t_n)=x|N(t_n)=y_n\} = P\{N(\infty)=y_n+x\}$$

or

$$P\{\bar{N}(t_n)=x|N(t_n)=y_n\} = \frac{a^{y_n+x}}{(y_n+x)!} e^{-a}$$

$$x=0,1,2,\ldots \qquad (15)$$

Also

$$E[\bar{N}(t_n)|N(t_n)=y_n] = a-y_n. \qquad (16)$$

This conditional distribution is important for deciding whether the software system under development can be released or not. The decision should be made based on the number of errors remaining in the software because it plays an important role in software reliability.

Reliability function. It can be shown that the reliability function, $R(t)$, after the last failure occurs at time s is given by

$$R(t) = e^{-a(e^{-bs}-e^{-b(s+t)})}. \qquad (17)$$

This is the conditional reliability function.

2.3 Estimation of Parameters

For the case under consideration the data is given in pairs (y_i,t_i), $i=1,2,\ldots,n$ where y_i is the number of software failure by time t_i. To obtain the estimates of parameters a and b of the model derived in Section 2.1, we proceed as follows.

Property (ii), along with properties (i), (iii) and (iv) of a NHPP, provides a complete statistical characterization for the NHPP so that the joint counting probability can be determined for any collection of times $0<t_1<t_2<\ldots<t_n$. That is,

$$P\{N(t_1)=y_1,N(t_2)=y_2,\ldots,N(t_n)=y_n\}$$
$$= \prod_{i=1}^{n} P\{N(t_i)-N(t_{i-1})=y_i-y_{i-1}\}$$
$$= \prod_{i=1}^{n} \frac{[m(t_i)-m(t_{i-1})]^{y_i-y_{i-1}}}{(y_i-y_{i-1})!} e^{-m(t_n)}. \qquad (18)$$

The likelihood function for given data (y_i,t_i), $i=1,2,\ldots,n$, is

$$L(a,b|y,t) = \prod_{i=1}^{n} \frac{\{a(e^{-bt_{i-1}} - e^{-bt_i})\}^{y_i - y_{i-1}}}{(y_i - y_{i-1})!}$$

$$e^{-a(1-e^{-bt_n})} \qquad (19)$$

Taking logarithm of (19), we get

$$\ell(a,b|\underline{y},\underline{t}) = \ell n L(a,b|\underline{y},\underline{t})$$

$$= \sum_{i=1}^{n} (y_i - y_{i-1}) \ell n a + \sum_{i=1}^{n} (y_i - y_{i-1}) \ell n (e^{-bt_i} - e^{-bt_{i-1}}) - a(1-e^{-bt_n}). \qquad (20)$$

Then, MLE's $\hat{a}$ and $\hat{b}$ must satisfy

$$\frac{\partial \ell}{\partial a} = 0, \qquad \frac{\partial \ell}{\partial b} = 0$$

or

$$a(1-e^{-bt_n}) = y_n \qquad (21)$$

and

$$at_n e^{-bt_n} = \Sigma \frac{(y_i - y_{i-1})(t_i e^{-bt_i} - t_{i-1} e^{-bt_{i-1}})}{e^{-bt_i} - e^{-bt_{i-1}}}. \qquad (22)$$

Equations (21) and (22) can be solved numerically to obtain the maximum likelihood estimates $\hat{a}$ and $\hat{b}$.

3. Software Error Data Analysis for a Large Scale Project

3.1 A Large Scale Software Project

The data under study has been taken from a large scale project reported in Thayer et al [16]. This project represents an initial delivery of a large command and control software package written in JOVIAL/J4. It consists of 115,346 total source statements and 249 routines. Some other characteristics of this project are summarized in Table 1. The error data used for this study is taken from the Software Problem Reports (SPR's) generated during the formal testing phase of this project. The majority of software errors were detected during Validation (Jun 1-Aug 12), Acceptance (Aug 13-Aug 24), and Integration phases (Aug 25-Oct 26). However, the operational data spanning a period of approximately one year (Oct 27-Nov 12) was also analyzed. The only time frame readily available from the data was the calendar day. The data also contain the mistakes by the operators and the "explanatory" errors, i.e. the fix is a change to a comment statement or the fix is not to a routine. These explanatory errors do or do not indicate the type of change. Therefore, the original data have to be restructured into four sets of data denoted by DS1 to DS4. The description and the total number of errors detected during the formal testing phase for each data set are given in Table 2.

In this paper the number of software errors detected during the formal testing is counted on a weekly basis. Also, for each data set the software errors detected during the first nine weeks are eliminated due to the fact that this represents the period of increasing number of software errors and we are interested in analyzing the software failures over the period when they are decreasing. Data about the SPR's for the 15 week period for the four cases (DS1 to DS4) can be obtained from references

3.2 Software Error Data Analysis

In this section we analyze the data sets DS1 to DS4 to develop time dependent models described above.

Mean value function. The simultaneous non-linear equations (21) and (22) are solved numerically for each data set to obtain the estimates $\hat{a}$ and $\hat{b}$. Thus, for data set DS1, the solution is $\hat{a}=1348$, $\hat{b}=0.124$ and the fitted mean value function is $1348\ (1-e^{-0.124t})$. This is also an estimate of the expected number of software errors detected by time t. A plot of the actual and the fitted values of the number of error detected during formal testing for this case is given in Figure 1. Also shown in this figure are the 90% upper and lower bounds for the N(t) process which can be computed from equation (9). Inspection of this figure indicates that the fit is very good. Estimates for other data sets are obtained similarly and are summarized in Table 3.

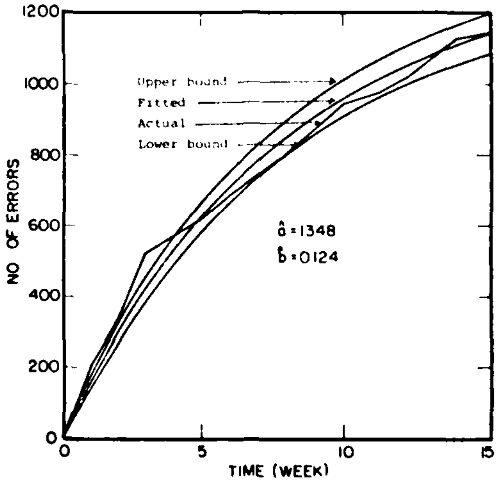


Fig. 1. Actual and fitted software errors and 90% bounds for the N(t) process for Data Set DS1

TABLE 1
Project Characteristics (TRW)

Size (Total source statement)	115,346
Number of routines	249
Language	JOVIAL/J4
Formal Requirements	To function level
Co-contractor	Yes
Subcontractor	No
Operating Mode	Batch
Formal Testing	Validation (6/1/73-8/12/73)
	Acceptance (8/13/73-8/24/73)
	Integration (8/25/73-10/26/73)
	Operational Demonstration (10/27/73-11/12/73)

TABLE 2
Description of the Data Set

Data Set	Description	Total Number of Errors	
		6/1/73-10/26/73 (24 weeks)	10/27/73-11/12/73 (22 weeks)
DS1	Original Data - TT - EX1 - EX2	2191	198
DS2	Original Data - TT - EX1	2621	263
DS3	Original Data - TT	4367	540
DS4	Original Data - TT - EX2	3937	475

TT represents the mistakes by the operators.
EX1 represents the explanatory errors which do not indicate what type of change (module, documentation, compool, data base) was involved.
EX2 represents the explanatory errors which indicate type of change.

TABLE 3
A Summary of Data Analyses

Quantity \ Data Set	DS1	DS2	DS3	DS4	F-11D
$\hat{a}$	1348	1823	3958	3466	107
$\hat{b}$	0.124	0.112	0.0768	0.0771	0.0367
$\mathrm{Var}(\hat{a})$	48.7	62.2	147.3	136.6	10.3
$\mathrm{Var}(\hat{b})$	0.00745	0.00643	0.00460	0.00492	0.00365
$\hat{\rho}_{a,b}$	-0.571	-0.648	-0.856	-0.855	-0.002
Estimated Number of Remaining Errors at the end of Integration Testing	210	340	1251	1084	0
Number of Errors Detected During Operational Demonstration Period	198	263	540	475	--

<u>Joint confidence region</u>. To obtain a $(1-\alpha)100\%$ joint confidence region for a and b we use the following approximation:

$$\ell(\hat{a},\hat{b}|\underset{\sim}{y},\underline{t})-\ell(a,b|\underset{\sim}{y},\underline{t}) = \frac{1}{2}\chi^2_{2;\alpha}. \quad (23)$$

From (23) and (20) we get

$$\sum_{i=1}^{n}(y_i-y_{i-1})\log a+\sum_{i=1}^{n}(y_i-y_{i-1})\log(e^{-bt_{i-1}}-e^{-bt_i})-a(1-e^{-bt_n}) = c \quad (24)$$

where

$$c = \ell(\hat{a},\hat{b}|\underset{\sim}{y}) - \frac{1}{2}\chi^2_{2;\alpha}$$

Confidence regions for desired values of α can be obtained by solving equation (24). For data set DS1, the joint confidence regions for a, b for α=.10, .25 and .50 are drawn in Figure 2.

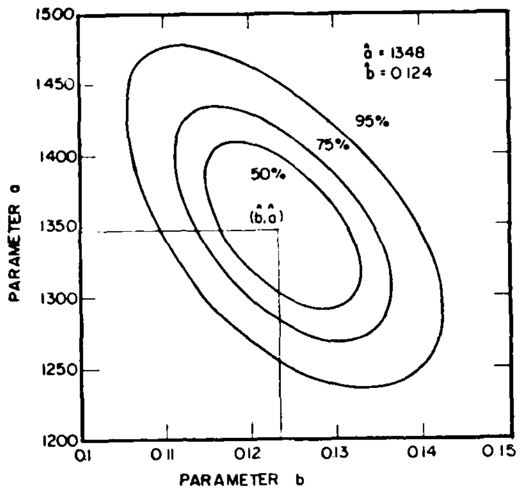


Fig. 2. Joint confidence regions for a and b for Data Set DS1

Asymptotic properties. For large n, the mle's $(\hat{a},\hat{b})$ follow a bivariate normal distribution (BVN):

$$\begin{pmatrix}\hat{a}\\ \hat{b}\end{pmatrix} \sim \text{BVN}\left(\begin{pmatrix}a\\ b\end{pmatrix} \quad \Sigma\right) \tag{25}$$

where the variance-covariance matrix Σ is given by

$$\Sigma = \begin{bmatrix} r_{aa} & r_{ab} \\ r_{ba} & r_{bb}\end{bmatrix}^{-1} \tag{26}$$

and

$$r_{ab} = -E\,\frac{\partial^2 \ell}{\partial a \partial b} \quad . \tag{27}$$

For data set DS1 the estimated variance-covariance matrix is

$$\hat{\Sigma} = \begin{bmatrix} 2368 & -0.2071 \\ -0.2071 & 5.554\times10^{-5}\end{bmatrix}. \tag{28}$$

From (28) the standard deviations of $\hat{a}$ and $\hat{b}$ are $\text{Var}(\hat{a})=48.7$, $\text{Var}(\hat{b})=0.00745$ and the estimated correlation coefficient is $\rho_{a,b}=0.571$.

Values of $\text{Var}(\hat{a})$, $\text{Var}(\hat{b})$ and $\hat{\rho}_{a,b}$ for data sets DS1 to DS4 are given in Table 3. Also given is the number of errors detected during the Operational Phase. By comparing the entries in the last two rows, we see that the predicted values are quite close to actual ones.

Expected number of remaining errors and confidence bounds. The expected number of remaining errors is computed from equation (13) for estimated values of a and b. Also, we can show that 100(1-α)% confidence bounds for $E\overline{N}(t)$ are given by

$$\{\hat{f}(a,b) \pm t_{n-2;\alpha/2}\ \hat{V}\{\hat{f}(a,b)\}\} \tag{29}$$

where

$$\hat{f}(a,b) = \hat{a}e^{-\hat{b}t}$$

$$\hat{V}\{\hat{f}(a,b)\} = \left(\frac{\partial f}{\partial a}\ \frac{\partial f}{\partial b}\right)\ \Sigma \left.\begin{pmatrix}\frac{\partial f}{\partial a}\\ \frac{\partial f}{\partial b}\end{pmatrix}\right|_{a=\hat{a},\ b=\hat{b}} \tag{30}$$

90% confidence bounds for $E\overline{N}(t)$ for data set DS1 are computed from the above equations and are shown in Figure 3. Also shown is a plot of the actual number of remaining errors during the 15 week period. From the figure we see that the actual errors fall within the 90% bounds.

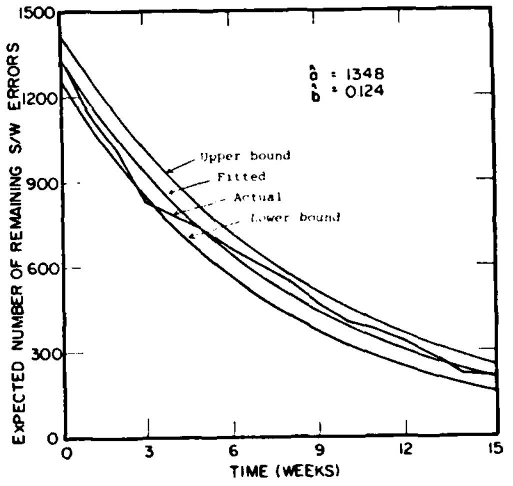


Fig. 3. 90% confidence bounds for the expected number of remaining software errors at time t for Data Set DS1

4. Software Error Data Analysis Based on CPU Time

In this section we discuss the analysis of software error data collected from a small project. The errors in this case are considered as a function of CPU times rather than the calendar time, as was the case for the data analyzed in Section 3.

4.1 Software Error Data

The error data, as reported by Wagoner [17], were collected from the computer program F-11D which obtains filtered navigation solutions from data taken by receiving equipment carried on an aircraft. This program is a data reduction program consisting of approximately 3000 4000 FORTRAN statements and required one month for development and checkout. The number of errors were counted with CPU (running) time. Wagoner [17] analyzed the data by a Weibull distribution with CPU time as the independent variable.

4.2 Data Analysis

The analysis procedure for this data set is similar to that of Section 3.2.

For the F-11D data given in [17], the estimates of a and b from (21) and (22) are $\hat{a}=107$ and $\hat{b}=0.0367$. The fitted mean value function is

$$\hat{m}(t) = 107(1-e^{-0.0367t}).$$

This is an estimate of the expected number of software errors detected by time t, where t is in CPU seconds. A plot of the actual and the fitted values of the number of errors detected during the development and checkout phases of F-11D program in CPU time is given in Figure 4. Also shown are the 90% upper and lower confidence bounds for the N(t) process. Inspection of this figure indicates that the fitted model satisfactory explains the actual error occurrence phenomenon.

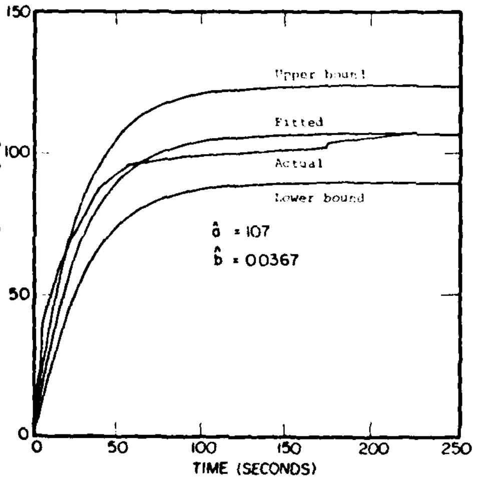


Fig. 4. Plots of the actual and fitted number of errors and confidence bounds versus CPU time

The joint confidence regions for a and b for α=0.10, 0.25 and 0.50 are drawn in Figure 5.

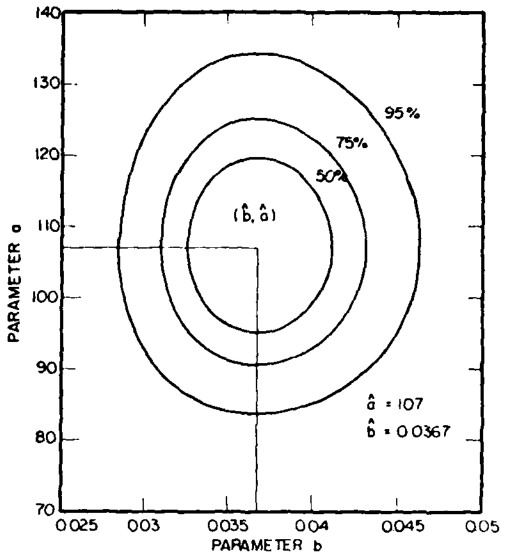


Fig. 5. Joint confidence regions for b and a for F-11D Data Set

The estimated variance-covariance matrix is

$$\hat{\Sigma} = \begin{bmatrix} 107 & -8.0\times10^{-5} \\ -8.0\times10^{-5} & 1.329\times10^{-5} \end{bmatrix},$$

and the estimated correlation coefficient is $\hat{\rho}_{a,b}=-0.002$.

The 90% confidence bounds for the expected number of remaining errors, $E\bar{N}(t)$, are shown in Figure 6. Also shown is a plot of the actual number of remaining errors during 250 CPU seconds period.

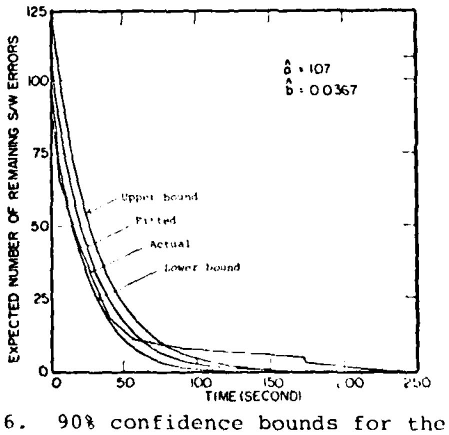


Fig. 6. 90% confidence bounds for the expected number of remaining software errors at time t for Data Set F-11D

5. Concluding Remarks

In this paper we have used a nonhomogeneous Poisson process to model the software error phenomenon in a two software systems, one large and one small. Based on a deterministic analysis of the failure process, we have derived the mean value function of this process. Various quantities of interest are derived to provide quantitative measures for software quality assessment. Estimation of parameters has been discussed and the results used to get fitted models for several data sets. Also, joint confidence bounds for the parameters are obtained. These can be used to obtain confidence bounds for the performance measures of the software system being analyzed.

6. References

[1] Akiyama, F., (1971), "An Example of Software Debugging," 1971 IFIP Congress, pp. TA-3-37 to TA-3-42.

[2] Cox, D. R. and Lewis, P. A., (1966), The Statistical Analysis of Series of Events, Methuen, London.

[3] Donelson, J. III, (1975), "Duane's Reliability Growth Model as a Nonhomogeneous Poisson Process," IDA Log No. HQ76-18012.

[4] Endres, A., (1975), "An Analysis of Errors and Their Causes in System Programs," Proceedings: 1975 International Conference on Reliable Software, pp. 327-336.

[5] Goel, A. L. and Okumoto, K., (1978), "An Imperfect Debugging Model for Reliability and Other Quantitative Measures of Software Systems," Technical Report, No. 78-1, Department of IE & OR, Syracuse University.

[6] Jelinski, J. and Moranda, P. B., (1972), "Software Reliability Research," 1972 International Symposium on Fault-Tolerant Computing, IEEE Computer Society.

[7] Littlewood, B. and Verrall, J. L., (1973), "A Bayesian Reliability Growth Model for Computer Software," Applied Statist., Vol. 22, pp. 332-346.

[8] Miyamoto, I., (1975), "Software Reliability in On-Line Real Time Environment," Proceedings: 1975 International Conference on Reliable Software, pp. 194-203.

[9] Musa, J. D., (1975), "A Theory of Software Reliability and its Application," IEEE Trans. on Software Engineering, Vol. SE-1, No. 3, pp. 312-327.

[10] Okumoto, K. and Goel, A. L., (1978), "Availability Analysis of Software Systems under Imperfect Maintenance," Technical Report No. 78-3, Department of IE & OR, Syracuse University.

[11] Roussas, G. G., (1973), A First Course in Mathematical Statistics, Addison-Wesley.

[12] Schick, G. J. and Wolverton, R. W., (1972), "Assessment of Software Reliability," McDonnell-Douglas Astronautics Company Paper WD1872.

[13] Schneidewind, N. J., (1975), "Analysis of Error Processes in Computer Software," Proceedings: 1975 International Conference on Reliable Software, pp. 337-346.

[14] Shooman, M. L., (1972), "Probabilistic Models for Software Reliability Prediction," Statistical Computer Performance Evaluation, pp. 485-502, Academic Press, New York .

[15] Sukert, A. N., (1977), "An Investigation of Software Reliability Models," Proc. 1977 R & M.

[16] Thayer, T. A. et al, (1976), "Software Reliability Study," TRW Defense and Space Systems Group, Final Technical Report, RADC-TR-76-238.

[17] Wagoner, W. L., (1973), "The Final Report on Software Reliability Measurement Study," Aerospace Report No. TOR-0074(4112)-1.

LAWS AND CONSERVATION IN LARGE-PROGRAM EVOLUTION

M.M. Lehman

Department of Computing and Control
Imperial College of Science and Technology
London SW7 2BZ

ABSTRACT

The paper analyzes the nature of the laws that have been identified by the work of Belady, Lehman and others. Program maintenance and evolution is planned, managed and implemented by people, yet the laws that govern the process are more akin to those of biology and even modern physics, then, as had been previously supposed, even more fuzzy than those that apply to economics and sociology.

After a brief discussion of the first four laws, highlighting the underlying phenomena and natural attributes, the paper concentrates on the fifth law. It shows what, how, and why it represents a conservation phenomenon, the conservation of familiarity.

INTRODUCTION

A recent paper [LEH78] discussed five laws of *large-program* evolution dynamics; laws that have emerged from the studies of Belady, Lehman and others over the last seven years as summarized in the bibliography of the above referenced paper. The main objective of the present contribution is to discuss one specific aspect of these laws, *conservation*. Some general introductory remarks are, however, desirable.

In the first place we should stress that the discussion here is limited to *large* programs defined by; "*a large-program is one that has been implemented or maintained by at least two independently managed groups*". Such a program will be the responsibility of an organization having two or more levels of management, will have the property of "variety" [BEL78] and will be outside the intellectual grasp of the individual. A program not satisfying the definition may possess one or another of the other properties and may display some or all of the characteristics of "*largeness*" [BEL78], but we do not consider them here.

PROGRAM EVOLUTION LAWS WITHIN THE SPECTRUM OF SYSTEMS OF LAWS

The evolution of software systems is clearly not a natural process governed by immutable laws of nature. Changes to a program are neither initated nor occur spontaneously. People do the work; amend or emend the requirement, the specification, the code, the documentation; repair the system; improve and enhance it. They do this in response to fault reports, user requests, business requirements, managerial directives or their own inspiration. Human thought and judgement plays a decisive role in the process that results from this continuing sequence of exogenous, seemingly stochastic, inputs.

Thus, we should not expect to discover laws of program evolution that yield the precision and predictability of the laws of physics [LEH77]. Any laws that emerge could reasonably have been expected to be even weaker than biological laws since the latter arise from observation of the collective behaviour of cellular systems which, whilst living, are, at least at the level of human understanding, non-intelligent. We should not even expect to observe behaviour that displays the regularity that has been abstracted into laws in the social and economic sciences, for example, the so-called "Law of Supply and Demand". After all the programming process is planned and controlled by an organizational and management structure that is sensitive and reactive to the demands, pressures, circumstances and contingencies of each moment. Thus, no regularity should be expected. Each action and event is surely determined by the needs of the moment. The process must surely be completely stochastic.

One of the first and most surprising, yet most fundamental, results of our observation and analysis of the dynamics of evolution of some eight programs, ranging over a wide spectrum of implementation and usage environments, has been that this is not so. Regularities, trends and patterns appear and dominate large program evolution. The common features and patterns of behaviour reflect common characteristics [BEL78] from which laws can be deduced; laws which, within the spectrum outlined, lie unexpectedly somewhere between the laws applying to biological organisms and those that emerge from the study of socio-economic systems. And these laws in turn can be used to create powerful, reliable and cost-effective life-cycle management tools.

THE UNDERLYING CAUSE FOR REGULARITY

Once the phenomenon has been recognized, the mechanisms underlying it are not difficult to understand. In the first instance, the program and its documentation in all their versions - the system - has a damping effect analogous to an ever-increasing inertial mass. As a totally unintelligent mechanism, the computer executes, and therefore impacts, its operating environment, precisely and only as the code in association with any input data, instructs. Good intentions, hopes of correctness, wishful thinking, even managerial edict cannot change the semantics of the code as written or its effect when executed. Nor can they affect the relationship between a specification and its implementation, or that between both of these and operational circumstances. That is the freedom of the designer, the implementor and the user to make changes or additions and to obtain the desired program behavior is increasingly constrained by existing code, documentation and past program application and behaviour. The code is unforgiving; there is no room for imprecision or logical error. Thus, any deviation leads to a need for corrective actions. The resultant feedback over the entire system process and organization increasingly causes the observed regularity.

These facts alone suffice to explain the consistency of the observations. Other factors merely strengthen the phenomenon. In particular large programs are, in general, created within large organizations and for large numbers of users; otherwise they could not be economically justified or maintained. Moreover, their very size and the complexity of both the program and the application for which it is intended means that decisions take time, sometimes considerable time, and large numbers of people to implement. The resultant delays provide exogenous pressures and endogenous opportunities for change. Thus the overall circumstance and environment acts like a filter to smooth out the global consequences of individual decisions, whilst also, paradoxically, adding the occasional stochastic disturbance. It also acts as a brake - economic and social - that inhibits or softens decisions that would have too drastic an impact. For example, large budgets can, in general, be neither suddenly terminated nor drastically increased. In practice they can only be changed by a fractional amount. Similarly a work force cannot be instantaneously retrained, relocated or dismissed; at best a task force can be sent in, and can cause a local perturbation.

In summary, large program creation and maintenance occurs in an environment with many levels of arbitration, smoothing and feedback correction that, in general, act to eliminate perturbations at the output. The existence of regularity and of laws abstracting that regularity becomes reasonable and understandable.

THE GROSS NATURE OF THE LAWS

The detailed instantaneous behaviour of the programming process and of the system that is the object of its activity, is the consequence of human decision and action. Specific individual events cannot therefore be predicted more precisely than can the specific acts of an individual. Any laws can thus only relate to the gross, statistical dynamics of a large program system over a period of time. But as such they find application in system prognosis, planning and project control. Equally (or even more importantly) they yield understanding that should permit improvement of the programming process and advance the development of software engineering science and practice.

FEEDBACK CONSEQUENCES OF INCREASING UNDERSTANDING OF THE PROCESS

Increasing understanding in turn raises another problem. To what extent can knowledge and understanding of the laws that regulate the programming process in an environment unaware or insensitive to their existence, be used to invalidate them, or for that matter, to perpetuate them by appropriate (or inappropriate) managerial responses? Space does not permit us to address this question in detail. We merely assert that the present laws reflect deeply rooted aspects of human and organizational behaviour. Associated with the mechanistic forces that define and control the automatic computational process, they are sufficiently fundamental to be treated as absolute, at least in our generation. As knowledge of them is permitted to impact the programming process, as programming technology advances, they may require restatement or revision, they may become irrelevant or obsolete. But for the time being, we must accept and learn to use, not to ignore, them.

THE LAWS

The First Law

We now comment briefly on the laws summarized in figure 1, so as to expose some of the more fundamental truths that they reflect. The laws have been fully discussed in earlier publications [LEH78 and bibliography].

The Law of Continuing Change arises from the fact that the world, in this case the computing environment, undergoes continuing change; all programs are models of some part, aspect or process of the world. They must therefore be changed to keep pace with the needs of a changing environment, or become progressively less relevant, less useful and less cost effective.

The Second Law

The Law of Increasing Complexity (an analogue or instance of the second law of thermodynamics) is a consequence of the fact that a system is changed to improve its capabilities and to do so in a cost-effective manner. Thus change objectives are expressed in terms of performance targets, system resources that will be required during execution, implementation resources, completion dates and so on. With multiple objectives it is impossible to optimize all simultaneously.

I. THE LAW OF CONTINUING CHANGE

A LARGE-PROGRAM THAT IS USED UNDERGOES CONTINUING CHANGE OR BECOMES PROGRESSIVELY LESS USEFUL. THE CHANGE OR DECAY PROCESS CONTINUES UNTIL IT IS JUDGED MORE COST-EFFECTIVE TO REPLACE THE SYSTEM WITH A RE-CREATED VERSION.

II. THE LAW OF INCREASING COMPLEXITY

AS A LARGE-PROGRAM IS CONTINUOUSLY CHANGED ITS COMPLEXITY, REFLECTING DETERIORATING STRUCTURE, INCREASES UNLESS WORK IS DONE TO MAINTAIN OR REDUCE IT.

III. THE FUNDAMENTAL LAW OF LARGE-PROGRAM EVOLUTION

THERE EXISTS A DYNAMICS OF LARGE-PROGRAM EVOLUTION WHICH CAUSES MEASURES OF GLOBAL PROJECT AND SYSTEM ATTRIBUTES TO BE CYCLICALLY SELF-REGULATING WITH STATISTICALLY DETERMINABLE TRENDS AND INVARIANCES.

IV. THE LAW OF INVARIANT WORK RATE

THE GLOBAL ACTIVITY RATE IN A LARGE PROGRAMMING PROJECT IS STATISTICALLY INVARIANT. (FOR EXAMPLE: NORMALLY DISTRIBUTED IN TIME WITH CONSTANT MEAN AND VARIANCE).

V. THE LAW OF CONSERVATION OF FAMILIARITY (PERCEIVED COMPLEXITY)

FOR RELIABLE, PLANNED, EVOLUTION, A LARGE-PROGRAM UNDERGOING CHANGE MUST BE MADE AVAILABLE FOR REGULAR USER EXECUTION (RELEASED) AT INTERVALS DETERMINED BY A SAFE MAXIMUM RELEASE CONTENT (CHANGED OR NEW) WHICH, IF EXCEEDED, CAUSES INTEGRATION, QUALITY AND USAGE PROBLEMS WITH TIME AND COST OVER-RUNS WHOSE CONSEQUENCES MAINTAIN THE AVERAGE INCREMENT OF GROWTH INVARIANT.

<u>FIGURE 1:</u> <u>THE FIVE LAWS OF LARGE-PROGRAM EVOLUTION</u>

Hence the completed project and system must represent a compromise that results from judgements and decisions taken during the implementation process, often on the basis of time, group or management local optimization. Structural maintenance, which is not often mentioned in project objectives since it yields no immediate or visible benefit, will inevitably suffer. Each change will thus degrade system structure a little more. The resultant continuing accumulation of gradual degradation, ultimately leads to the point where the system can no longer be cost-effectively maintained and enhanced unless and until a clean-up is undertaken.

<u>The Third Law</u>

The Fundamental Law of Large-Program evolution was previously called the *Law of Statistically Smooth Growth* [LEH78]. It expresses the observation we have already made above that large-program evolution is not a purely stochastic process that, at each instance, reflects the decisions and actions of the people in the environment in which it is maintained and in which it is used. The Law states that, at least in the current state of the art, there exists a dynamics whose characteristics are determined during conception and the early life of the system, the process and the organization that maintains them. The characteristics of this dynamics increasingly determines the gross trends of the maintenance and enhancement process. Feedback effects are an inherent factor in the self-stabilizing control process that evolves. Hence cyclic effects, not necessarily with a pure period, emerge.

<u>The Fourth Law</u>

The Law of Invariant Work Rate is assumed to reflect a conservation property. The quantity or quality being conserved has however not been clearly identified. The Law is believed to be a consequence of the fact that, in general, human organizations seek, and seek to maintain, stability or, more accurately, stable growth. As suggested above, sudden changes in, for example, staffing, budget allocations, manufacturing levels, product types are avoided, are, in general, not possible. A variety of managerial, union and governmental checks, balances and controls ensure overall progress to the ever changing, ever distant objective of the organization; or its collapse. The Law also, in a sense, reflects the organizational response to the limitation which, we shall show, underlies the fifth law.

Thus with hindsight it becomes clear that the discovery of some derivative of an activity measure that is invariant, or better, statistically invariant (for example, normally distributed in time with constant mean and variance) could have been anticipated. What is not really understood (except vaguely as reflecting the limitations of absorbtive capacity) is why in large-program maintenance projects, measures of work input rate should be the quantity to display such invariance. But the fact remains that for all the systems observed, the count of modules changed (handled) or changes made per unit of time, as averaged over each release interval, has been statistically invariant over the period of observation.

The Fifth Law

The Law of Conservation of Familiarity (Perceived Complexity) was previously referred to [LEH78] as the *Law of Incremental Growth Limits.* Its discovery was based on data from three of the systems observed, each of which was made available to users on a release basis. In each case the incremental growth of the program varied widely from release to release. But the average over a relatively large number of releases remained remarkably constant. That is a high-growth release would tend to be followed by one with little or no growth, or even by a system shrinkage. Or two releases, each of about average growth, would be followed by one with only slight growth. Moreover releases for which the net growth exceeded about twice the average, proved to be minor, or major, disasters (depending on the degree of excess) with poor performance, poor reliability, high fault rates, cost and time over-runs. This evidence suggests that initial release quality is a non-linear function of the release incremental growth. From a more complete phenomenological analysis, along the lines outlined below, and for which a mathematical model needs to be constructed, we hypothesise that the quality is exponentially related to the magnitude of the changes implemented in the release.

The phenomenon was detected at a very early stage of our evolution dynamics studies, was featured in our earliest models [BEL71], and has been applied as a planning and control parameter for a number of years. The full explanation, however, has only recently become apparent.

The release process was originally identified as a stabilization mechanism [BEL71]. Once a large-program is in general use its code and documentation are normally in a state of flux. A fault is fixed locally, perhaps fixed differently or not at all in other installations. Minor or major changes, local adaptations, are made. Code is changed without a corresponding change to documentation. Documentation is changed to correspond to observed behaviour without a full and detailed analysis of the precise semantics of the code within the context of the total system and under all possible environmental conditions. Only at the moment of release does there exist an authoritative version of the program, the code and its documentation. Even this may include multiple versions of modules say, for clearly defined alternative situations.

For an old release each implementor, each tester, each salesman, each user will be familiar with the version of the program with which he has been associated. This familiarity will have bred, not contempt, but a certain degree of relaxation, of ability to work with the program in order to accomplish specific objectives. The program will be manipulated or used without (apparent) need for concentrated thought. External perception of its intrinsic complexity will be at a minimum. In the limit the program may be said to be approaching zero *perceived complexity* for people working consistently on or with it.

As changes are introduced, as the new release is gradually created and as it becomes available, new and unfamiliar code appears. The program behaves differently in execution, in its interaction with and impact on the operational environment. Even the pagination in the previously familiar documentation has changed and any need for reference entails a major search. The system has become uncomfortably unfamiliar, the degree of unfamiliarity depending on the extent and inaptitude of the change (See below.).

A major intellectual effort is now required by each person involved before any completely successful and cost-effective interaction with the new system can occur. The system has suddenly become complex. Its perceived complexity is high.

Even those who participated in the preparation of the new release will normally have been involved directly with only a small part of the change, a small portion of the system. They too can only learn to understand the new system in its totality from the moment of its completion. Moreover until the complete system is available all acquisition of knowledge and understanding of the changes and of the new system, must be based on *reading* of code and documentation text, or on partial execution of system components on test cases or system models. Only with final integration does the full executable program become available. We shall suggest that when the release content exceeds some critical amount only *operational* experience with the *complete* system can bring or restore the degree of knowledge and familiarity, the global viewpoint, that is essential for the cost-effective maintenance, enhancement and exploitation of the large program.

Thus, in general, at the moment of release or shortly before that time a major learning effort will begin and this will involve all those associated with the system, not just the users. All changes and additions must be identified, understood and <u>experienced</u>, their significance within the operational context of the total system, appreciated. Once this has been done the old degree of comfort with the system will return, its perceived complexity is once again zero, the level of familiarity has been restored.

Clearly the amount of hard work that must be involved to achieve this, the intellectual effort required, depends amongst other factors, on the attitudes of people, on the organization, on the number and magnitude of changes introduced. Moreover, as already observed, this dependence must be at least quadratic, probably exponential. For changes to the system interact with one another. Changes implemented in the same release, that is in the same time interval, must each be understood not only in themselves, but also in the context of all the others, of the unchanged parts of the system and of past and future applications. The source of the drastic growth in difficulty in restoring the familiarity required of and with the program if it is to be once again correctly and efficiently maintained and exploited, is clear. Figure 2 is intended to suggest '"difficulty" versus magnitude of release content' relationship. The axis of the curve are not calibrated since at present neither concept nor suitable measures are well defined. But their relevance to the Fifth Law requires us to analyze the nature and consequence of these concepts. Incidentally, the concept of "difficulty" introduced here clearly relates to that of Norden and Putnam [SLC77]. But the precise relationship has not yet been established.

Before proceeding with the analysis one brief remark should be made. The Fifth Law as now formulated talks about release content and its magnitude (previously incremental growth). The content certainly includes new or changed code, but must also involve deleted code, new or changed function and new or changed documentation. For example, the observations that inspired the law were measured in terms of modular growth and we have in fact consistently found module-based measures in terms of modular growth and we have in fact consistently found module-based measures to be more accurate and useful than those based on instruction counts. But there is at present no clear or agreed measure of release content, the magnitude of a change. It is not even clear that the concept possesses a metric. It must be left to the future to identify or define measures and to provide an improved formulation of the Fifth Law. Meanwhile we must clarify the concepts and increase understanding of, at least, the phenomenology, thereby we will provide a basis for ultimate formalization.

With this clarification we may now proceed with the analysis. Everyone's ability to master a new or changed object is limited, though people clearly differ in their capability to absorb the new knowledge, to achieve full understanding of the changed program. Thus the impact of changes will undoubtedly vary from person to person according to many factors that will include, but are not limited to, their learning ability and absorptive capacity. But for a given large-program in its given environment and where many people are inevitably involved, the delays that are incurred, the mistakes that are made, the destructuring that occurs before full familiarity is restored, the direct and indirect cost of familiarization, depends on the average ability of all the people involved. After all, while the above-average person will regain mastery more quickly, make fewer mistakes, achieve a temporary advantage (which will probably cause him to be promoted or moved out of the project fairly quickly), the below-average person will fall behind, will perhaps lose contact, make more mistakes, do more damage. He may well be re-assigned to a less demanding role with less impact on the project or even fired. But the damage will have been done and with hiring policies established make up of the project, the average level will remain at best, unchanged; more probably decline [LEH64].

For different organizations, different systems, different structures, different methodologies, different processes, the average level will, of course, be different. This implies that any models will contain exogenous variables. Thus it establishes a potential for improving the level in any given circumstance once the phenomenon, the organization and the programming process are understood.

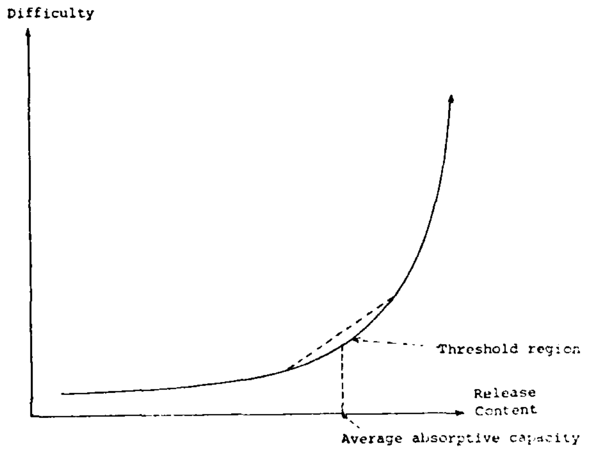


FIGURE 2: DIFFICULTY - RELEASE CONTENT RELATIONSHIP

Given the above insight into the growth of difficulty in understanding and working with a system as the release content increases, and its consequences, as a result of feedback that slows down both utilization and further evolution as system structure deteriorates, that the number of faults increases, documentation lags and performance declines, we are now in a position to appreciate the Fifth Law.

If the release content, the magnitude of the change and/or the incremental growth, is less than some threshold region (Figure 2) the integration and operational installation of the new system should be fairly straightforward. No major problems should be experienced in mastering the new release; it may well be that the change may be absorbed, familiarity restored without actual system operation exposure.

When the release content lies in a threshold region which may not be precisely delineable, quality, performance, completion and installation problems are to be expected. Slippage and cost overrun may occur. A subsequent, low growth, release will be required to clean up the system and restore it to a state that permits further cost-effective evolution.

Finally if a release whose content exceeds the threshold region is attempted serious problems will be encountered. Slippage and cost over-run will occur unless the plans take account of the greatly increased difficulties that will be experienced. If not properly planned it may lead to the effective collapse of the system or, as we have observed in at least two instances, to an effect that we have termed system *fission*. Since only release of the system to end users and to the developers provides full exposure, even when adequate resources and time has been provided, such a release will still require to be followed by a restoration or clean up release.

The Fifth Law abstracts these observations, adding an additional factor, that of the emergence of invariant average incremental growth or content. The latter is also a consequence of the additional exogenous pressure for accelerated functional growth or content. The latter is also a consequence that is a characteristic feature of large-program applications and, in general, of organizational environments.

FINAL WORD

One last word should be added. The first recognition of the laws discussed was based entirely on an examination and analysis of data from some very large programs. However, once they are formulated the laws must be examined in their own right to achieve the transition from phenomenology to science. These laws of large-program development and evolution are now beginning to be understood in this way. They are seen to express very basic attributes of computing, of the programming development, maintenance and usage processes, of programs themselves, and of the organizations and environments in which these activities are carried out.

Once this interpretation of the laws in terms of more fundamental phenomena has been achieved we must re-examine the old data and examine new information in the light of the laws *as understood*. Deviations must be explained and interpreted. Contradictions may require re-formulation and re-interpretation of a law or even its rejection. There is, of course, nothing new in these comments. They form the very basis of the scientific method. They are added here to assert the belief that the laws as formulated have been substantiated by experience and by experimental data to the point where they can stand in their own right until evidence and developing insight and understanding demands their change.

REFERENCES

[BEL71] L.A. Belady and M.M. Lehman: "Programming System Dynamics or the Meta-Dynamics of Systems in Maintenance and Growth": IBM Research Report, RC3546, September 1971, pp.30.

[BEL78] L.A. Belady and M.M. Lehman: "Characteristics of Large Systems": Part I, Chapter 3, in "Research Directions in Software Technology". Sponsored by the Tri-Services Committee of the DOD. Proc. of the Conference on Research Directions in Software Technology, October 10-12, 1977, Brown U., Providence, R.I. and MIT Press 1978.

[LEH69] M.M. Lehman: "Mediocraty in Middle Management", unpublished manuscript.

[LEH77] M.M. Lehman: "Human Thought and Action as an Ingredient of System Behaviour": in "Encyclopaedia of Ignorance", Ed. Duncan and Weston-Smith, Pergamon Press, 1977, pp.347-354.

[LEH78] M.M. Lehman: "Laws of Evolution Dynamics - Rules and Tools for Programming Management": Infotech State of the Art Conference, "Why Software Projects Fail", April 1978. To be published in Conference Proceedings 1978.

[SLC77] In "Software Phenomenology - Working Papers of the (First) S.L.C.M. Workshop", Airlie, VA, August 1977. Published by ISRAD/AIRMICS, Computer Systems Command, U.S. Army, Fort Belvoir, VA, Dec. 1977.

VALIDATION OF A SOFTWARE RELIABILITY MODEL

Bev Littlewood

Mathematics Department, City University,
St. John Street, London EC1V 4PB, England

Summary

The paper reports some preliminary results from an attempt to use real data to validate the software reliability model proposed by Littlewood and Verrall [1]. A goodness-of-fit test is employed to compare actual times-to-failure with the distributions of times-to-failure predicted by the model. The test shows the model to fit with remarkable fidelity. It is suggested that this is evidence in support of the author's earlier comments [2], [3].

1. Introduction

In recent papers [2, 3] I have criticised some of the assumptions underlying much work on software reliability measurement. In particular, I have suggested that some authors [3, 4, 5, 6, 7] make assumptions about the relationship between dynamic performance measures (failure rate, distribution of future time-to-failure, etc.) and static measures (number of remaining bugs) which are extremely implausible. I have argued, also, that some of the peculiar properties of software (e.g. lack of natural degradation, uniqueness of each item) might result in much of the conventional reliability theory, created for hardware systems, being inappropriate. It is my contention that the model developed jointly by John Verrall and myself avoids some of these pitfalls - albeit at the expense of other disadvantages. The results reported here are the beginning of an attempt to validate this model using real-life data. It is intended to examine more closely in future work the points raised in [2] and [3], but these preliminary results appear to lend encouraging support.

2. Description of the calculations

Our model is described in references [1], [8] and [9], where details of the parameter estimation procedures can be found. Briefly, we assume (in common with most workers in this field) that the time between (i-1)th and ith failures of the program has (conditionally) the exponential pdf:

$$\text{pdf}(t_i|\lambda_i) = \lambda_i e^{-\lambda_i t_i} \tag{1}$$

In order to model the reliability growth which takes place as a result of the bug-fixing attempts, we treat $\{\lambda_i\}$ as a sequence of random variables with pdfs:

$$\text{pdf}(\lambda_i|\alpha, \psi(i)) = \frac{\psi(i)\{\psi(i)\lambda_i\}^{\alpha-1}\exp\{-\psi(i)\lambda_i\}}{\Gamma(\alpha)} \quad \text{for } \lambda_i > 0, \tag{2}$$

$$= 0, \text{ otherwise.}$$

It is easy to see that the unconditional distribution of T_i is not exponential.

The unknown quantities in the model are α and $\psi(i)$. It is via the second of these that the model reflects past and future changes in reliability, so this growth function will be of particular practical importance. We suggest [1] that a choice of parametric family be made for this function, then the estimation problem concerns the parameters of the function together with α.

Our original hope, when first working on the model, was that the growth function parametric family might be known *a priori*. This now seems unlikely, so we propose that several families be tried and that which best fits the data be used. It is fortunate that the inference procedure adopted allows a comparison to be made between the families (maximum likelihood techniques, for example, do not enable a choice to be made between models in this way).

When the parameters of the model have been estimated, distributions of future times-to-failure can be calculated. From these it is possible to calculate any reliability measures of interest, such as failure rate, reliability function. Since the purpose of this study is validation of the model, I have concentrated on the percentage points of the predicted time-to-failure distributions.

Table 1 lists the 136 observations: these are the successive execution times (in seconds), each terminating with a software failure. Notice the great variability: a feature of software reliability data which must be reflected by any good model.

Figure 1 is an example of part of the output from the program which performs the calculations. In this case, the first 35 observations were used in the calculation, and the growth function being used was linear: $\psi(i) = \beta_1 + \beta_2 i$. There are thus three unknown parameters: α, β_1, β_2, which are estimated using the 35 data points. The routine which estimates these β's (α is taken care of via a Bayesian analysis) operates by searching in the 2-dimensional β-space to find those values which best fit the 35 data points observed so far. The program allows this search to be started at two different points: the two results should be quite close as a check on convergence. An objective measure of the quality of the fit of the growth function is given by the value of the goodness-of-fit statistic, WSTAT: this should be small. Based on estimates of the parameters utilising the 35 failure times so far observed, the program then calculates and displays percentage points of the next 20 predicted times-to-failure. Thus 19.65 is the 10% point, 57.11 the 25% point, 158.94 the 50% point (median), etc., of the next (36th) time-to-failure. The number in parentheses gives the position on the predicted distribution of the actual observation: it is these numbers which are used to validate the model. Thus in this case, the actual 36th observation (65 seconds) lies at the 0.276 (27.6%) point of the distribution predicted from 35 failure times.

The calculations have been performed successively upon the first 30, 35, 40, ... , 120, 125 observations in the series. Since there is no a priori reason for choosing a particular growth function, the calculations have all been carried out on both

$$\psi(i) = \beta_1 + \beta_2 i$$

and

$$\psi(i) = \beta_1 + \beta_2 i^2.$$

The measure of goodness-of-fit (WSTAT in Fig. 1) is used to choose between these for each calculation. Table 2 shows that the linear growth function is superior except for 40, 45, 105 observations.

3. The results

It should be emphasised that the objective of these calculations was to measure the quality of the predicted time-to-failure distributions. This is a more stringent test than procedures such as comparing predicted mttf's with average times-to-failure. In fact, a secondary objective was to see whether my suspicion of the non-existence of mttf for software could be tested [2, 3].

It can easily be seen that if the model is a good one, the numbers in parentheses in the output (see Fig. 1) can be regarded as a random sample from a uniform distribution on (0, 1). This observation underlies the results shown in Figures 2, 3, 4.

Fig. 2 shows the quality of predictions 1 to 5 steps ahead using the linear growth function. Here 100 actual observations are compared with the 100 predicted distributions based on calculations using successively 30, 35, ... 125 observations. As can be seen from the Kolmogorov-Smirnov test, the fit is extraordinarily good.

Even the quadratic growth function, which Table 2 shows to be inferior to the linear one, gives an excellent fit (see Fig. 3).

In Figure 4 a test is carried out of longer term prediction. Here projections are made relating to the distributions 16 to 20 failures in the future, using the linear growth function. The quality of these results is, again, remarkably good.

It is particularly noticeable, in Figures 2 and 4, that the fit is at its best in the right hand tail of the distribution. Since it is precisely here that our model differs from the simpler exponential models, by having long right-hand tails in the predicted time-to-failure distributions (causing infinite mttf), this seems evidence in favour of the controversial allegations of my earlier papers [2, 3].

4. Conclusion

Although the results shown here relate to only one set of data, they are encouraging in the support they give to our model. In fact the quality of fit between predictions and observations is better than I expected when I embarked upon this validation exercise. This work will continue, and it is hoped to analyse more data shortly, but already there is the beginnings of a case in support of the arguments given in references [2] and [3].

The program which performs these calculations is still being developed, and it is hoped to publish a version of it eventually. In the meantime, if anyone would like a listing of the current version, please let me know. Alternatively, I would be very happy to have the opportunity of analysing any software failure data which readers may have.

Acknowledgement

I am very grateful to John Musa for providing the data upon which this paper is based, and for many helpful suggestions and comments over the last couple of years. His forbearance in the face of my own awful software engineering practices is also much appreciated.

References

[1] LITTLEWOOD,B. and VERRALL,J.L., "A Bayesian reliability growth model for computer software" Applied Statistics (J. Royal Statist. Soc., Series C) Vol. 22, 3, 1973, pp. 332-346.

[2] LITTLEWOOD,B.,"Software reliability measurement - some criticisms and suggestions", in Software Phenomenology, working papers of 1st SLCM workshop,1977, pp. 473-488.

[3] LITTLEWOOD,B., "How to measure software reliability, and how not to...", Proceedings of 3rd International Conference on Software Engineering,IEEE, 1978, pp. 37-45.

[4] JELINSKI,Z. and MORANDA,P.B., "Software

reliability research", in Statistical Computer Performance Evaluation, Ed.: W. Freiberger. New York: Academic, 1972, pp. 465-484.

[5] SHOOMAN,M., "Operational testing and software reliability estimation during program development", in Record, 1973 IEEE Symposium on Computer Software Reliability, NY, NY, 1973, pp. 51-57.

[6] MUSA,J.D., "A theory of software reliability and its application", IEEE Trans. on SE, SE-1, 3, 1975, pp. 312-327.

[7] GOEL,A.L. and OKUMOTO,K., "An imperfect debugging model for reliability and other quantitative measures of software systems", Tech. Report 78-1, April 1978, Dept. of Ind. Eng. and O. R., Syracuse U., NY 13210.

[8] LITTLEWOOD,B. and VERRALL,J.L.,"A Bayesian reliability growth model for computer software", same source as [5], pp. 70-76.

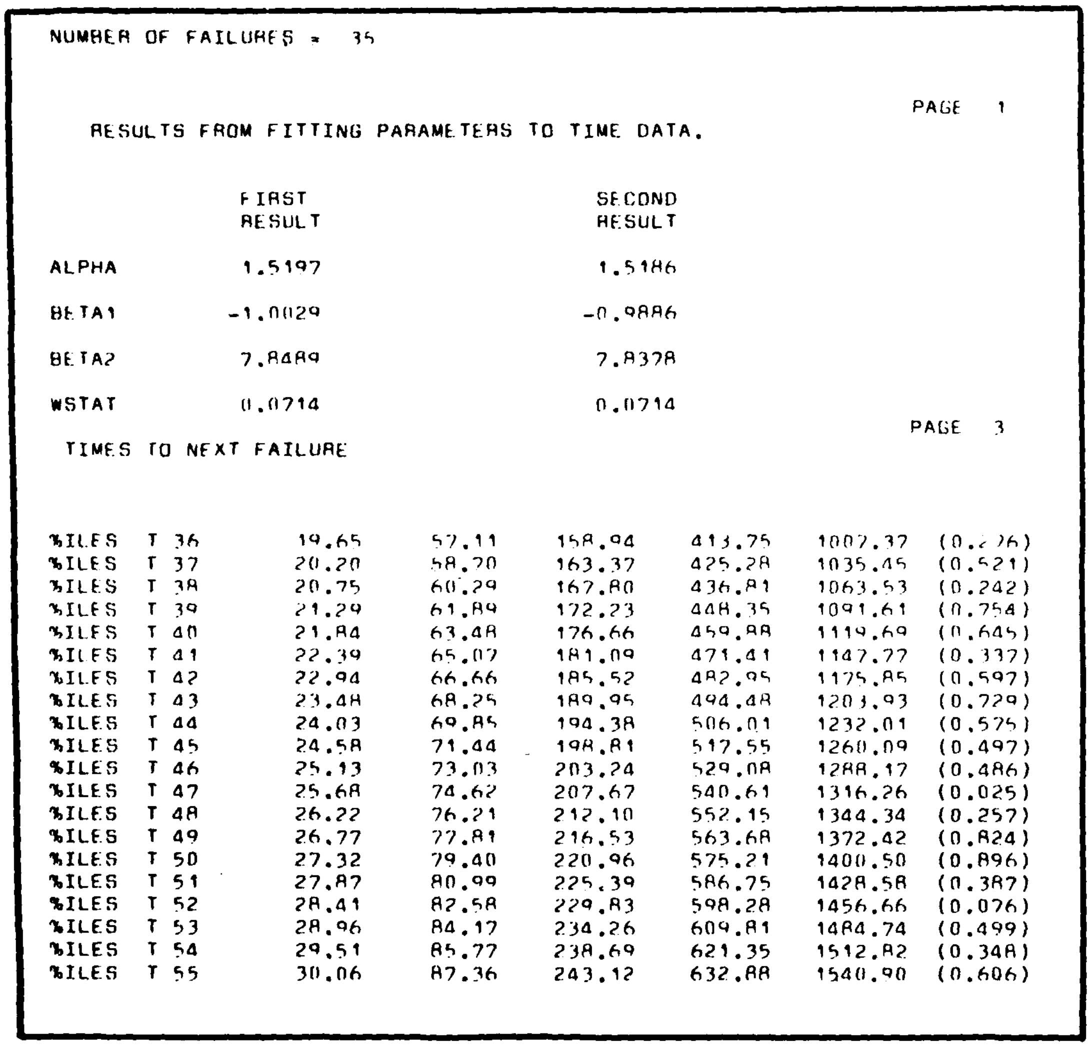
[9] LITTLEWOOD,B. and VERRALL,J.L., "A Bayesian reliability model with a stochastically monotone failure rate", IEEE Trans. on Rel., R-23, 2, 1974, pp. 108-114.

Number of observations calculations based upon	W Statistic (WSTAT)	
	$\psi(i) = \beta_1+\beta_2 i$	$\psi(i) = \beta_1 i+\beta_2 i^2$
30	.0259	.0327
35	.0714	.0717
40	.0617	.0589
45	.0571	.0567
50	.0524	.0553
55	.0427	.0511
60	.0427	.0587
65	.0595	.0822
70	.0918	.1088
75	.0842	.1028
80.0	.0706	.0885
85	.0687	.0886
90	.0761	.0874
95	.0755	.0802
100	.0780	.0800
105	.1021	.1016
110	.0999	.1001
115	.0817	.0860
120	.1027	.1066
125	.1243	.1267

Table 2. Comparison of the two growth functions for the series of calculations. It can be seen that the linear function provides a better fit except in the cases of calculations based upon 40, 45, and 105 observations.

```
3. 30. 113. 81. 115.
9. 2. 91. 112. 15.
138. 50. 77. 24. 108.
88. 670. 120. 26. 114.
325. 55. 242. 68. 422.
180. 10. 1146. 600. 15.
36. 4. 0. 8. 227.
65. 176. 58. 457. 300.
97. 263. 452. 255. 197.
193. 6. 79. 816. 1351.
148. 21. 233. 134. 357.
193. 236. 31. 369. 748.
0. 232. 330. 365. 1222.
543. 10. 16. 529. 379.
44. 129. 810. 290. 300.
529. 281. 160. 828. 1011.
445. 296. 1755. 1064. 1783.
860. 983. 707. 33. 868.
724. 2323. 2930. 1461. 843.
12. 261. 1800. 865. 1435.
30. 143. 109. 0. 3110.
1247. 943. 700. 875. 245.
729. 1897. 447. 386. 446.
122. 990. 948. 1082. 22.
75. 482. 5509. 100. 10.
1071. 371. 790. 6150. 3321.
1045. 648. 5485. 1160. 1864.
4116.
```

Table 1. The 136 successive failure times upon which the calculations are based. Notice the great variability of the data, and the obvious improvement in reliability as time passes.



```
NUMBER OF FAILURES =  35

                                                                 PAGE   1
   RESULTS FROM FITTING PARAMETERS TO TIME DATA.


                    FIRST                      SECOND
                    RESULT                     RESULT

ALPHA               1.5197                     1.5186

BETA1              -1.0029                    -0.9886

BETA2               7.8489                     7.8378

WSTAT               0.0714                     0.0714
                                                                 PAGE   3
 TIMES TO NEXT FAILURE



%ILES   T 36       19.65      57.11      158.94     413.75     1007.37  (0.276)
%ILES   T 37       20.20      58.70      163.37     425.28     1035.45  (0.521)
%ILES   T 38       20.75      60.29      167.80     436.81     1063.53  (0.242)
%ILES   T 39       21.29      61.89      172.23     448.35     1091.61  (0.754)
%ILES   T 40       21.84      63.48      176.66     459.88     1119.69  (0.645)
%ILES   T 41       22.39      65.07      181.09     471.41     1147.77  (0.337)
%ILES   T 42       22.94      66.66      185.52     482.95     1175.85  (0.597)
%ILES   T 43       23.48      68.25      189.95     494.48     1203.93  (0.729)
%ILES   T 44       24.03      69.85      194.38     506.01     1232.01  (0.575)
%ILES   T 45       24.58      71.44      198.81     517.55     1260.09  (0.497)
%ILES   T 46       25.13      73.03      203.24     529.08     1288.17  (0.486)
%ILES   T 47       25.68      74.62      207.67     540.61     1316.26  (0.025)
%ILES   T 48       26.22      76.21      212.10     552.15     1344.34  (0.257)
%ILES   T 49       26.77      77.81      216.53     563.68     1372.42  (0.824)
%ILES   T 50       27.32      79.40      220.96     575.21     1400.50  (0.896)
%ILES   T 51       27.87      80.99      225.39     586.75     1428.58  (0.387)
%ILES   T 52       28.41      82.58      229.83     598.28     1456.66  (0.076)
%ILES   T 53       28.96      84.17      234.26     609.81     1484.74  (0.499)
%ILES   T 54       29.51      85.77      238.69     621.35     1512.82  (0.348)
%ILES   T 55       30.06      87.36      243.12     632.88     1540.90  (0.606)
```

Figure 1. Example of output from the program which performs the calculations described in the text of the paper. WSTAT is the value of the goodness-of-fit statistic, used in the optimisation routine, at its minimum: i.e. at the point in β-space which will be used as the estimate in the predictions. It is the value of this statistic which enables a comparison to be made between different growth functions: the best growth function is that which has smallest WSTAT.

The percentiles given here are, from left to right, 10%, 25%, 50% (median), 75%, 90%. The last column shows the position of each actual observation on the distribution which is predicted for it based, here, on the first 35 observations.

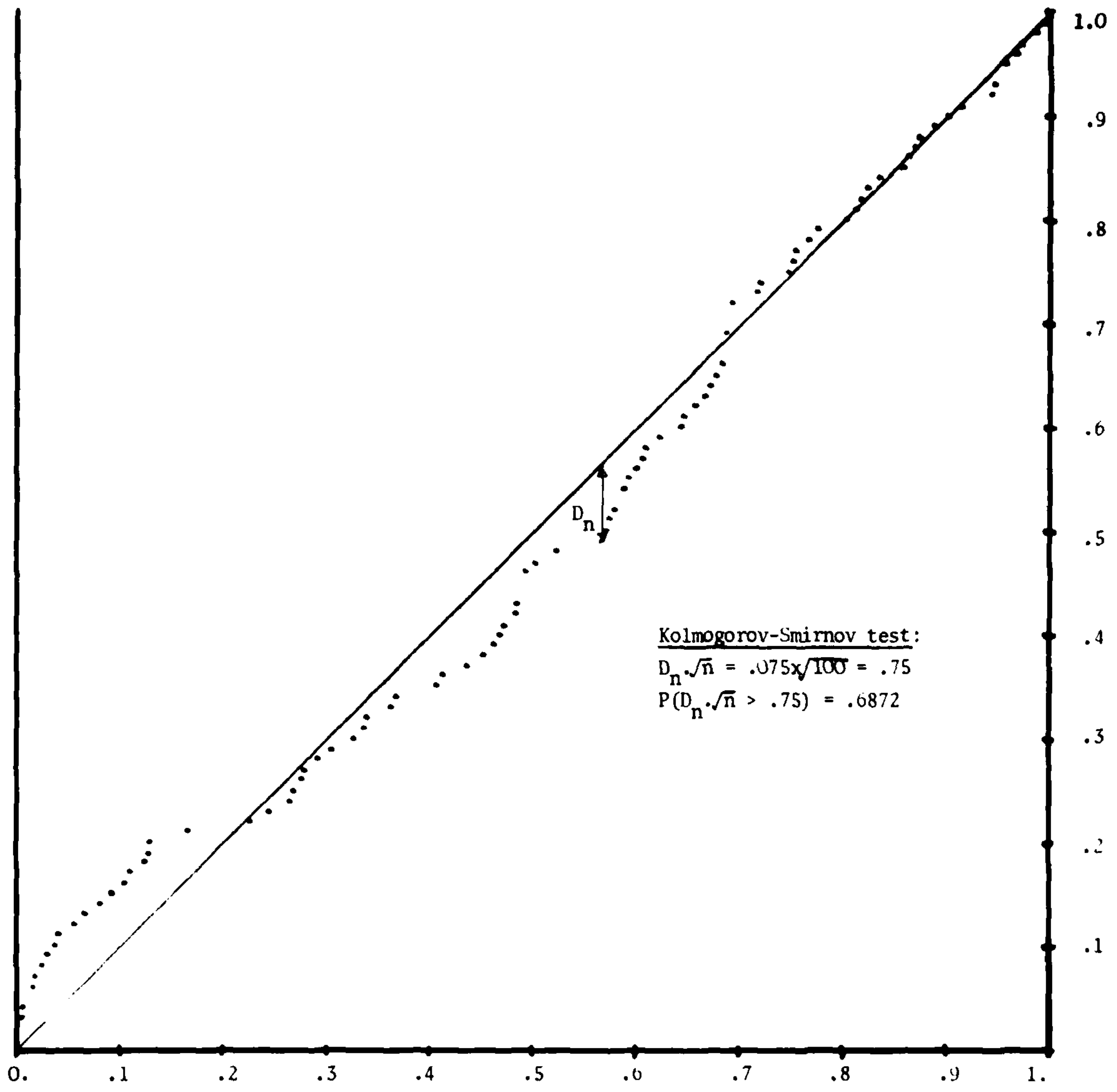


Figure 2. Test of fit between actual observations and predicted distributions, using a linear growth function. There are 100 observations used; the prediction is 1 to 5 steps ahead.

The fit is extremely good, as may be seen from the K-S value of .6872. Notice particularly the closeness of the points in the right-hand portion of the plot to the line.

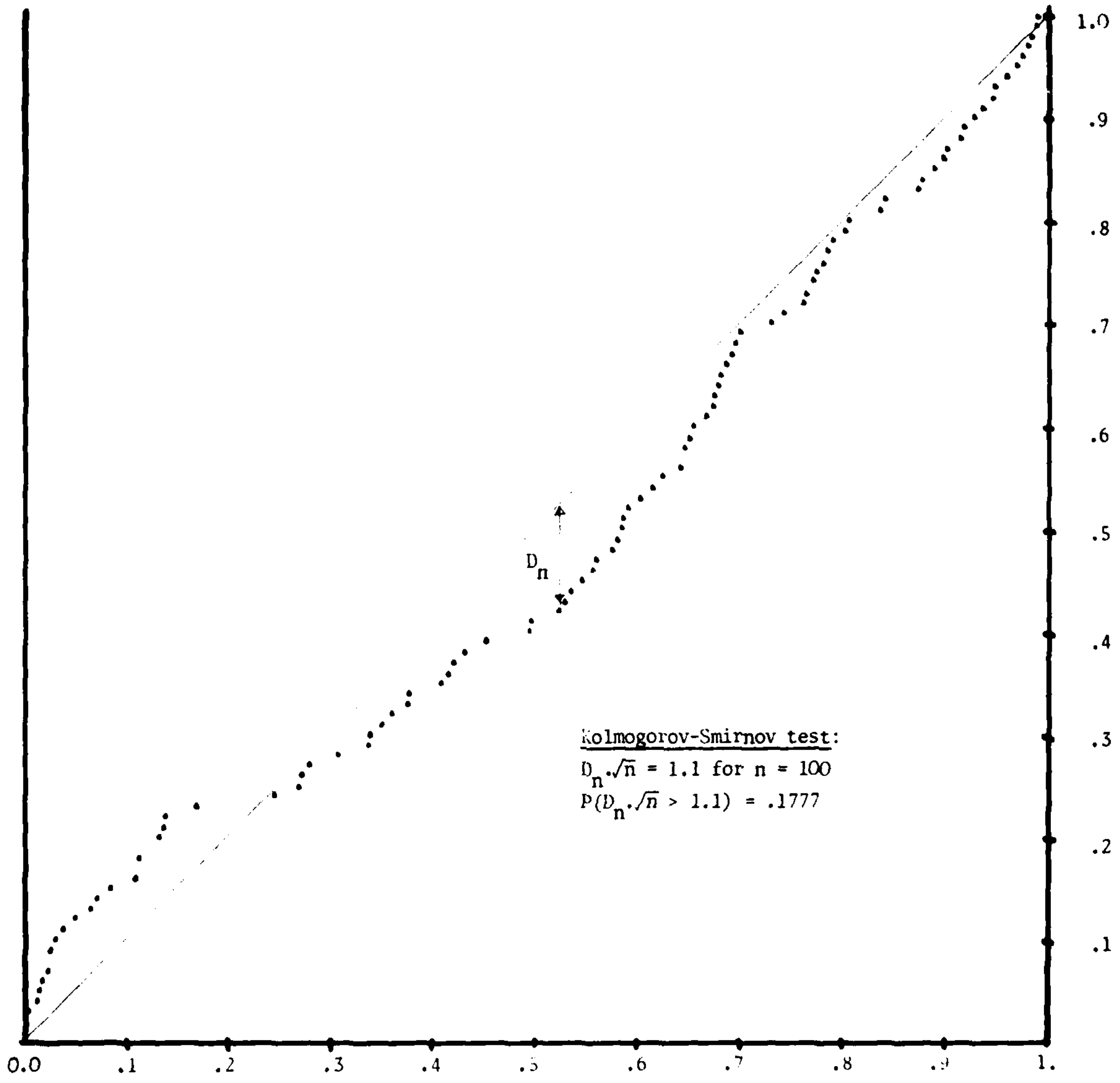


Figure 3. Quadratic growth function, prediction 1 to 5 steps ahead. Although the quadratic does not fit as well as the linear function (see Table 2), the predictions are again very good. The K-S value is much smaller than for the linear case, but is still not significant at the 10% level.

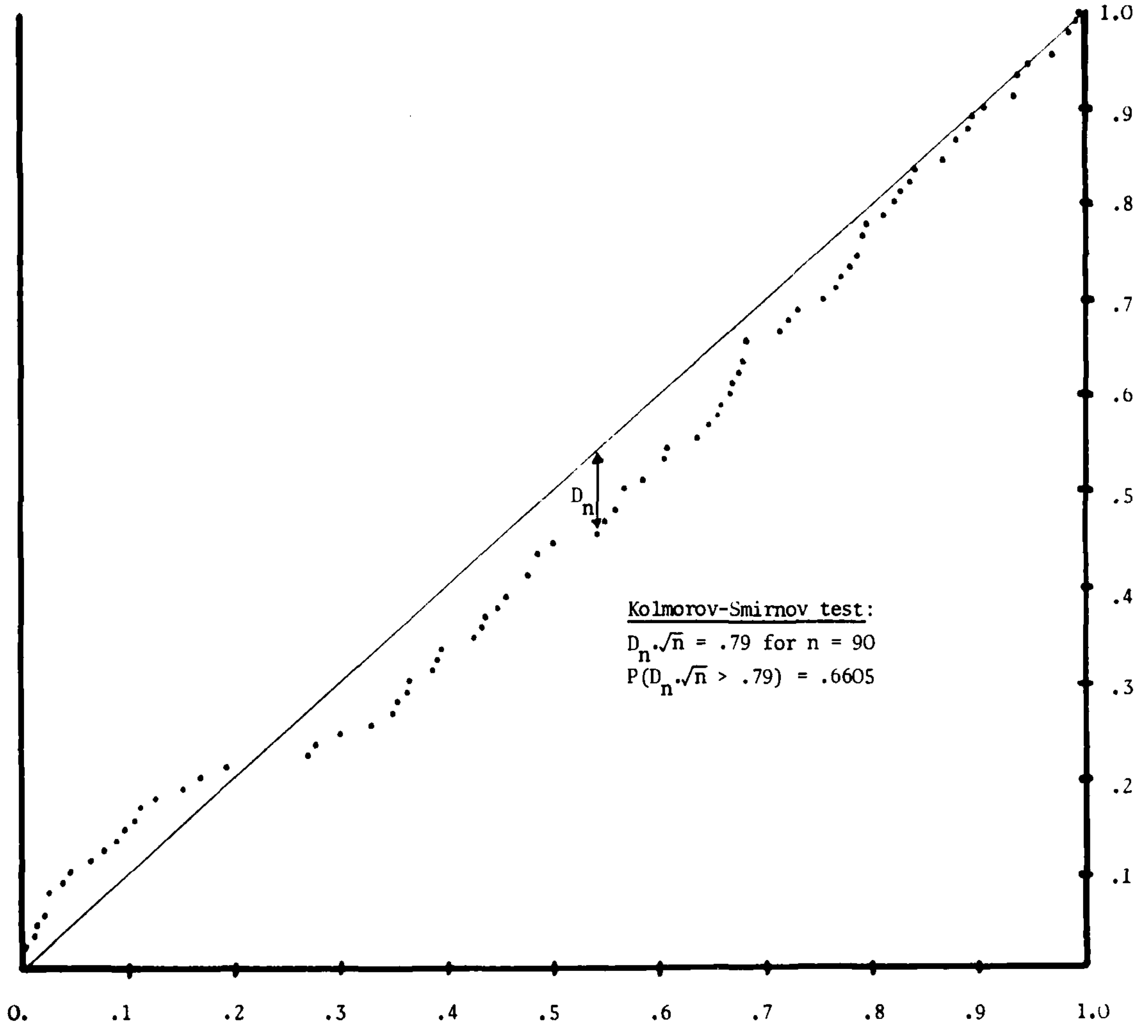


Figure 4. Linear growth function, longer range prediction: 16 to 20 steps ahead. This time only 90 observations could be used because of the length of the data vector. Again the fit is extremely good, as seen from the K-S statistic. Good fit in right-hand tail of distributions again.

PROGRESS IN SOFTWARE RELIABILITY MEASUREMENT

John D. Musa

Bell Telephone Laboratories
Whippany, N. J. 07981

ABSTRACT

This paper summarizes progress made in the past year in the application of the execution time theory [1] of software reliability. It also discusses a continuing mutual critical examination of the Littlewood model [2] and the author's execution time theory.

Introduction

Work associated with the execution time theory of software reliability in the last year falls into four principal areas:

(a) collection of data for model verification,

(b) accumulation of experience in application of the theory to project development, with resultant refinements,

(c) application of the theory to measurement of computation center software, involving service monitoring and change control, and

(d) investigation of execution time model simulation as an aid to management decisions.

In addition to the foregoing work, substantial effort was expended in mutual critical examination of the Littlewood model and the author's execution time theory, resulting in disposition of some of the issues that had been raised at the first SLCM Workshop and sharpening others.

Data Collection

Failure interval data has now been collected for ten software development projects and four operational software systems. Resource expenditure data has been collected for four development projects. This data is being analyzed with regard to the execution time model of software reliability and its assumptions to determine how well or poorly the model fits reality.

Project Development Experience

The execution time theory has been and is currently being applied to various development projects. Considerable experience has been accrued in handling particular circumstances related to each project [3]. The accumulation and organization of this experience is continuing. Some of the particular problems that have been addressed are:

(a) handling design changes,

(b) specifics of parameter estimation, and

(c) details of program parameter reestimation.

Current work includes investigation of methods for improving predictions made for projects that employ sequential integration and testing.

Measurement of Computation Center Software

The execution time theory was recently applied to operating system software in a large general purpose computation center over a period of 15 months [4]. Mean-time-to-failure (MTTF) trends of software components were shown to correlate very well with periods of system degradation or improvement. The trends can be monitored as a basis for controlling system modification (modifications are suspended when the MTTF is below an acceptable level), reassigning maintenance personnel among software components, and determining when to make cutovers to new releases or regressions to old releases.

Simulation as an Aid to Management Decisions

Some experimentation was conducted with simulation of the execution time model as an aid in making management decisions [5]. Parameters of the model that were associated with different managerial options were selected, and simulation was used to determine the effects on schedules and costs.

For example, for one project, the predicted completion date was considered to be too late and simulations were run to determine the effects of adding various resources to the project and of reducing the MTTF objective. For this particular project at the particular point in time studied, adding people or working overtime were found to have no effect. Providing more computer resources did permit the completion date to be advanced. Reducing the MTTF objective had only a moderate impact on the completion date up to a point; after that point, the impact was appreciable.

Littlewood and Execution Time Models

B. Littlewood and the author have been engaged in a continuing critical dialogue concerning each other's software reliability models. This was initially stimulated by a more general critique of software reliability models published by Littlewood [6]. This critique was updated after some of the discussions took place [7]. The following is the author's understanding of the current status of this dialogue.

Many issues have been resolved or have vanished as a result of clearer understanding (this does not mean that these same issues may not exist for other software reliability models). Both models are oriented toward failures and operational reliability. We agree that it is important to be able to estimate the number of errors because this quantity is related to the repair effort that will be required and hence, project schedules. We are both in agreement that execution time rather than calendar time is the key metric related to software reliability. Both models assume an exponential distribution of failure intervals.

In the Littlewood model, the failure rate parameter of the exponential distribution is assumed to have a gamma distribution, one of whose parameters accounts for reliability growth. The growth parameter is some function (usually in two subparameters) of the sequential failure number. In the author's model, the exponential distribution failure rate parameter is a fixed (essentially two subparameter) function of the sequential failure number. The two subparameters, initial MTTF and total failures expected, have readily understood physical interpretations.

Both models view the repair process as being statistical in nature. In the Littlewood model, this uncertainty is exhibited in the exponential distribution failure rate parameter itself having a distribution. This distribution can be reflected in the distribution of time to n-th failure from the present, whose percentiles can be estimated. In the author's model, the uncertainty is reflected in confidence intervals estimated for the subparameters. Littlewood's subparameters are estimated in themselves but confidence intervals are not determined for them.

Littlewood's approach requires that the form of the repair function be selected from all the many various possibilities and the subparameter values be determined that minimize a goodness of fit statistic (search over a surface is required). The author's approach uses maximum likelihood estimation to determine the subparameter values and their confidence intervals. Littlewood's approach is somewhat more general, at a cost of one or two orders of magnitude greater computing requirements. The author questions whether the added generality is worth the cost; the author's approach may well be sufficiently accurate for most practical purposes.

The generality of Littlewood's approach quickly leads to analytical difficulties; it is not computationally practical to predict the execution time required to reach a specified failure interval objective. It therefore is not possible to predict project completion date, which is of vital importance to managers. The author's approach does yield these predictions.

Littlewood and the author differ on a fundamental point, which perhaps is based on the characteristics of a mathematical/statistical versus an engineering approach to the problem (i.e., focusing on rigor versus focusing on utility). Littlewood points out that the author's model is dependent (due to the implicit assumption that the MTTF exists) on the postulate that it is certain that a program is imperfect or will eventually fail. Based on experience as a software manager and on writing many programs, the author believes that this is an excellent model of reality. The rare cases when this postulate may not be true do not justify the added complexity and the added computation that the Littlewood model involve.

Although the author prefers the execution time model because he feels it is a more understandable, useful, and economic engineering and managerial tool, the Littlewood model's somewhat greater generality and rigor are attractive. Consequently, studies are in progress to attempt to relate the two models. Hopefully, the Littlewood model can be tested on data from projects where the author's model was applied. Attempts are being made by both Littlewood and the author to speed up the minimization process the Littlewood model uses. Perhaps these studies will result in refinements to both models. At any rate, their similarities and differences and advantages and disadvantages should come into sharper focus.

References

[1] J. D. Musa, "A Theory of Software Reliability and Its Application," IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, pp. 312-327, September 1975.

[2] B. Littlewood and J. L. Verrall, "A Bayesian Reliability Growth Model for Computer Software," J. Roy. Statist. Soc. (Series C, Applied Statistics), Vol. 22, No. 3, pp. 332-346, 1973.

[3] J. D. Musa, "Software Reliability Measurement," in Software Phenomenology - Working Papers of the Software Life Cycle Management Workshop, Airlie, Virginia, pp. 427-452, August 21-23, 1977.

[4] Patricia A. Hamilton and John D. Musa, "Measuring Reliability of Computation Center Software," in Proc. 3rd Int. Conf. Software Engineering, Atlanta, Ga., pp. 29-36, May 10-12, 1978.

[5] John D. Musa, "The Use of Software Reliability Measures in Project Management," paper submitted to 2nd Int. Computer Software and Applications Conference, Chicago, Nov. 13-16, 1978.

[6] B. Littlewood, "Software Reliability Measurement: Some Criticisms and Suggestions," in Software Phenomenology - Working Papers of the Software Life Cycle Management Workshop, Airlie, Virginia, pp. 473-488, August 21-24, 1977.

[7] B. Littlewood, "How to Measure Software Reliability, and How Not to," in Proc. 3rd. Int. Conf. Software Engineering, Atlanta, Ga., pp. 37-45, May 10-12, 1978.

THE WORK BREAKDOWN STRUCTURE IN SOFTWARE PROJECT MANAGEMENT

Robert C. Tausworthe

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, California 91103

The Work Breakdown Structure (WBS) is a vehicle for breaking an engineering project down into subproject, tasks, subtasks, work packages, etc. It is an important planning tool which links objectives with resources and activities in a logical framework. It becomes an important status monitor during the actual implementation as the completions of subtasks are measured against the project plan. Whereas the WBS has been widely used in many other engineering applications, it has seemingly only rarely been formally applied to software projects for various reasons. Recent successes with software project WBSs, however, have clearly indicated that the technique can be applied, and have shown the benefits of such a tool in management of these projects.

This paper advocates and summarizes the use of the WBS in software implementation projects. The paper also identifies some of the problems people have had generating software WBSs, and the need for standard checklists of items to be included.

Introduction

If one were to be given the task of writing a program in which the target language instruction set was not entended to be executed by some dumb computer, but, instead, by intelligent human beings, then that programmer might be thought to have an easier job than his colleagues who write their programs for machines. However, a little reflection will show that his job is much more difficult for a number of reasons, among which are ambiguities in the English language and a multitude of human factors.[1] However, such a program, often named the PLAN, is an essential part of almost every industrial project slated for success.

One of the difficulties in writing this program is the supplying of enough detail so as to be executable without ambiguity by those programmed. Another is getting the right controls into the program so that the programees perform as stated in the PLAN. Still another is making the plan complete, having all contingencies covered and a proper response to each supplied. And one final problem of note here is making the plan bug-free, or reliable, so that once execution starts, if everything proceeds according to the PLAN, there is no need to deviate.

Programmers well-schooled in modern techniques[2] would approach the writing of this PLAN in a structured way, using top-down design methodolology modular development, stepwise refinement, hierarchic layering of detail, structurally sound constructions, and semantically definite documentation. Such an approach would tend to bring a measure of organization to the PLAN, understandability to its documentation, and reliability to its execution. If created in this way, the resulting format of the PLAN work tasks would have the attributes of what is known in the engineering industry as a "Work Breakdown Structure".[3]

The Work Breakdown Structure (WBS) is an enumeration of all work activities in hierarchic refinements of detail, which orgainzes work to be done into short, manageable tasks with quantifiable inputs, outputs, schedules, and assigned responsibilities. It may be used for project budgeting of time and resources down to the individual task level, and, later, as a basis for progress reporting relative to meaningful management milestones. A software management plan based on a WBS contains the necessary tools to estimate costs and schedules accurately and to provide visibility and control during production.

Such a plan may be structured to evaluate technical accomplishments on the basis of task and activity progress. Schedules and PERT/CPM[4] networks may be built upon technical activities in terms of task milestones (i.e., accomplishments, outputs, and other quantifiable work elements). Projected versus actual task progress can be reviewed by technical audit and by progress reviews on a regular (say monthly or bi-weekly) basis. Formal Project Design Reviews are major check points in this measurement system.

But knowing modern programming theory does little good if one does not also have the programming experience to apply it to. Similarly, the knowledge of what a WBS is, what its goals are, what its benefits are, and what its structure is supposed to be like, does not necessarily instruct one in how to apply that knowledge toward developing a WBS for his or her own project.

In the coming sections of this paper, I shall

*The work reported in this paper was carried out at the Jet Propulsion Laboratory of the California Institute of Technology under contract NAS 7-100, sponsored by the National Aeronautics and Space Administration.

review some of the characteristics and benefits of the WBS, and then discuss how these can be developed and applied in software implementation projects. I will orient this material principally toward new-software production tasks, although many of the concepts will be applicable also to continuing maintenance and operations tasks, as well.

The Work Breakdown Structure

The goals assumed here for generating the WBS are to identify work tasks, needed resources, implementation constraints, etc. to that level of detail which yields a sought-for accuracy in the original PLAN, and to provide the means for early calibration of this accuracy and corrective replanning, if required, during the actual implementation.

How refined should this WBS be? Let me answer this question by showing how the WBS and schedule projection accuracy are interrelated.

If a project has identified a certain number of equi-effort milestones to be achieved during the course of implementation, then the mere number of milestones achieved by a certain date is an indicator of the progress toward that goal. A graph of accumulated milestones as a function of time, sometimes called a "rate chart", permits certain predictions to be made about the future completion date rather handily and with quantifiable accuracy, especially if the milestones are chosen properly.

Let it be supposed that it is known a priori, as a result of generating the WBS, that a project will be completed after M milestones have been met. These milestones correspond to all the tasks which have to be accomplished, and once accomplished, are accomplished forever (i.e., some later activity does not re-open an already completed task; if such is the case, it can be accommodated by making M larger to include all such milestones as separate events). The number M, of course, may not be precisely known from the first, and any uncertainty in M is certainly going to affect the estimated completion date accuracy. Such uncertainties can be factored in as secondary effects later, as needed for refinement of accuracy.

Now, let it be further supposed that it has been possible to refine the overall task into these M milestones in such a way that each task is believed to require about the same amount of effort and duration to accomplish. Viewed at regular intervals (e.g., bi-weekly or monthly), a plot of the cumulative numbers of milestones reported as having been completed should rise linearly[5] until project completion.

More quantitatively, let m be the average number of tasks actually completed during each reporting period, and let σ be the standard deviation of the actual number of milestones completed each reporting period about the mean value (the values of m and σ are presumed to be constant over the project duration). The value of m is a reflection of the team average productivity and σ is a measure of the ability estimate their production rate. Both are attestations to team effectiveness -- first, in their ability to produce, and second, in the ability to create a work plan which adequately accounts for their time.

The project should require M/m reporting periods to complete, which time, of course, should not depend on whether a WBS was made or not (I am discounting, in this discussion, whether WBS generation increases or decreases productivity). Thus, M/m should be a constant value, relatively speaking. If M is made large, tasks are smaller and shorter, so proportionately more of them are completed each reporting period. The project schedule will, in fact, assume some productivity, or mean accomplishment rate, but an actual performance value will generally be unknown until progress can be monitored for some period of time.

But while the numbers M and σ may not affect the team productivity, they do directly influence the effectiveness with which a project can monitor its progress and predict its future accomplishments. Generation of a WBS, of course, gives (or estimates) the parameter M. Monitoring the completion of milestones provides estimates for m and σ. From these, projections of the end date and calculations for the accuracy of this prediction can be made. Based on such information, the project can then divert or reallocate resources to take corrective action, should progress not be deemed suitable.

In this simplified model, a best straight-line fit through the cumulative milestone progress over the first r reports (of an expected R=M/m reports) at regular ΔT intervals will predict the time required to reach the final milestone. It will also provide an estimate of m and σ. The normalized predicted completion date may be expected to deviate from the projected value (as a one-sigma event) by no more than[5]

$$\sigma_M \doteq 1.48\, \sigma_1\, (R/rM)^{\frac{1}{2}}$$

within first-order effects. The value $\sigma_1 = \sigma/m^{\frac{1}{2}}$ represents the normalized variance of an individual task milestone (it is limited to values of less than unity in the underlying model).

The bound permits the specification of WBS characteristics which enable accurate early predictions of future progress. High overall accuracy depends on a combination of low σ and large M. One may compensate for inaccurate appraisals of productivity only by generating a very detailed WBS!

As an example, suppose that a 10% end-date prediction accuracy is required by the end of the first quarter (r/R = .25) of a project. Then the tradeoff figure is $M/\sigma_1{}^2 = 876$. Hence, if the WBS is highly uncertain (σ=1), that WBS should contain 876 equi-duration milestones! If the project is confident that it can hold more closely to its average productivity (and has most contingencies provided for), with a $\sigma_1=0.5$, then it needs only about 220 milestones. A one-man-year project with bi-weekly reporting, one milestone per report (26 milestones in all) must demonstrate a $\sigma_1=0.17$ level of task prediction accuracy!

It is therefore both necessary and important to generate a detailed WBS rather carefully, and to monitor milestone achievements relative to this WBS very faithfully, if accuracy in predicting the future progress of a project is of great importance.

Reasonable Schedule Accuracy

A project engineer on a 2-year, 10-man task may perhaps be able to manage as many as 876 subtasks, each formally assigned and reported on. That amounts to about one subtask completion per week from each of the 9 workers. But the generation of the descriptions for the 876 tasks will require considerable effort. Moreover, it is unlikely that such a detailed plan would have a σ also as large as one week; if the project engineer has the ability to break the work accurately into 876 week-long subtasks, he or she can probably also estimate the task deviations well within a week.

The ability of the project engineer (or planning staff) to generate a clear and accurate WBS will determine the level to which the WBS must be taken. Greater accuracy of the work breakdown definition produces greater understanding and clarity of the actions necessary to complete task objectives. If the work is understood, readily identified, and achievable as discerned, the confidence of reaching the objectives is high. Thus, the further the subtask descriptions become refined, the better the estimator is able to assess the individual subtask durations and uncertainties. Refinement ceases when the sought-for M/σ_M^2 is reached.

Practically speaking, a work-plan with tasks shorter than one week in duration will usually require too much planning and management overhead to be worthwhile. On the other hand, a work plan with tasks longer than one or two weeks will probably suffer from a large σ_1. Thus, a breakdown into one- or two-week subtasks is probably the most reasonable target for planning purposes.

A work-year consists of about 47 actual weeks of work (excluding vacation, holidays, sick-leave, etc). Therefore, a project of w workers can reasonably accommodate only about 47w/d tasks per year (including management tasks) of duration d weeks each; spread over y years, the total number of milestones can reach M=47wy/d, so that the practical accuracy limit one may reasonably expect at the one-quarter point in a project (r/R=.25) is about

$$\sigma_M \le 0.432\ \sigma_1(d/wy)^{1/2}$$

Note that accuracy is related to the total man-year effort in a project, other things being equal. A 3-man-year project completing 1 task per manweek can expect to have $\sigma_M \le 0.216\ \sigma_1$; with a $\sigma_1=0.4$ ($\pm$2 days per weekly task) The end-date estimation accuracy is within 10%.

Generating The WBS

There is no mystery about making a WBS. People do it all the time, although they seldom call the result a WBS. Most of the things we do, in fact, are probably first organized in our heads, and for small undertakings, most of the time that works out well. For more complex undertakings, especially those involving other people, it becomes necessary to plan, organize, document, and review more formally.

The general algorithm for generating a WBS is even fairly simple to state. It goes something like this:

1. Start with the project statement of work, and put this TASK on top of the "working stack".
2. Consider the TASK at the top of the working stack. Define technical performance objectives, end-item objectives, reliability and quality objectives, schedule constraints, and other factors, as appropriate; inputs and materials required for starting the task; accomplishments and outputs which signal the completion of the task; known precedent tasks or milestones; known interfacing tasks; and resources required, if known. Determine whether this task can be accomplished within the duration (or cost) accuracy goal.
3. If the goal is achieved, skip to the next step; otherwise, partition the current TASK into a small number of comprehensive component subtasks. Include interfacing tasks and tasks whose output is a decision regarding substructuring of other subtasks. Mark the current TASK as a "milestone", pull its description off the working stack, push it onto the "finished stack", and push each of the subtask descriptions onto the working stack.
4. Repeat from step 2 until the working stack is empty.
5. Sequence through all items from the "finished" stack and accumulate durations (costs) into the proper milestones.

The steps in this algorithm are not always simple to perform, nor can they always be done correctly the first time, nor without sometimes referring to items already put into the "finished" list. The process is one of creation, and thus it requires judgement, experience, identification of alternatives, tradeoffs, decisions, and iteration. For as the project statement of work is refined, eventually the implementaltion of the program itself appears as one of the subtasks to be refined. When this subtask is detailed into component parts, the work descriptions begin to follow the influences of the program architecture, organizational matters, chronological constraints, work locations, and "whatever makes sense".

Therefore, the formation of the WBS, the detailed planning, and the architectural design activity are all mutually supportive. The architecture indicates how to structure the tasks, and the WBS goals tell when the architectural phase of activity has proceeded far enough. Scheduling makes use of the WBS as a tool and in turn influences the WBS generation by resolving resource conflicts.

But there are many subtasks in a software project which are not connected with the architecture directly, such as requirements analysis, project administration and management, and preparations for demonstration and delivery. The structure of these subtasks, being independent of the program architecture, can be made fairly standard within a given organization for all software productions. However, since there is no automatic or closed-loop means to guarantee whether all the planning factors that need to be put into the WBS actually get put into it, a *standard WBS checklist* can be a significant boon to proper software project planning, to decrease the likelihood of something "dropping through the cracks".

A Standard WBS Checklist

Previous DSN experience[6] at The Jet Propulsion Laboratory with WBS methodology has permitted moderately large software implementation projects to detect schedule maladies and to control project completions within about 6% of originally scheduled dates and costs. The WBSs were formed by individuals with extensive software experience, overseen by an expert manager. None of the software individuals had ever made a WBS before, and the manager had never tried one on a software project. Together, with much travail, they assembled ad hoc items into a workable system.

A candidate standard WBS outline and checklist is currently being assembled and evaluated within The Deep Space Network (DSN) at The Jet Propulsion Laboratory. This Standard WBS checklist includes many factors gained from previous successes and contains items to avert some of the identified shortcomings. Table I shows the upper-level structure of this WBS checklist. Detailed task descriptions are also in process of documentation and evaluation. A short application guidebook is planned, to instruct cognizant individuals in the method, approach, and practice.

Such a checklist and guidebook, together with useful automated WBS entry, update, processing, and report generation aids impose standards on software projects that are intended to facilitate the project management activity and make it more effective. Initial scheduling and downstream rescheduling of subtasks are aided by a WBS database that contains precedence relationships, durations, costs, resource requirements, resource availability and similar constraints on each subtask. PERT and critical-path methods (CPM) are applied directly to the WBS database, resulting in a preliminary schedule. Alterations of this schedule are then effected by editing the WBS via additional constraints recorded into the database. Actual production progress is measured by marking milestone completions, which are then plotted into a rate chart and all significant milestones are projected to a best-estimate completion date.

Problems

The Work Breakdown Structure is a well known, effective project engineering tool. It has not been applied to software projects as often as it has to hardware and construction, probably because the planning and architectural design tasks in software have not always been sufficiently integrated as to be mutually supportive, because all of the management, support, and miscellaneous tasks were seldom fully identifiable and detailable during the planning phase, because separation of work into manageable packets quite often requires design decisions properly a part of the *detailed design phase*, because a basis for estimating subtask durations, costs, and other constraints has not existed or been known; and because software managers have not been trained in WBS methodology. Modern software engineering studies of phenomenology and methodology are beginning to close the gaps, however.

The existence of useful tools and methods does not assure their acceptance, nor does their acceptance insure project success. Plans and controls are essential project aids, but unfortunately, they also do not guarantee success. The WBS is a planning, monitor, and control tool whose potential for successful application within a software project has been demonstrated. However, further researches and demonstrations are necessary before a WBS-oriented software planning and control methodology and system are as well integrated into the software industry as structured programming has only recently become. Fortunately, many organizations and individuals are sensitive enough to the software management crisis of past years that inroads are being worked on[7]

Happily, the solutions will almost certainly not be unique, but will range over limits which accommodate management and programming styles, organizational structures, levels of skill, areas of expertise, cost and need-date constraints, and human and technical factors.

Table I

SOFTWARE IMPLEMENTATION PROJECT

Detailed Work Breakdown Structure

Outline

1. ANALYZE SOFTWARE REQUIREMENTS
   - .1 Understand functional and software requirements
   - .2 Identify missing vague, ambiguous, and conflicting requirements
   - .3 Clarify stated requirements
   - .4 Verify that stated requirements fulfill requestor's goals
   - .5 Assess technology for supplying required software
   - .6 Propose alternate requirements or capability
   - .7 Document revised requirements
2. DEVELOP SOFTWARE ARCHITECTURE
   - .1 Determine architectural approach
   - .2 Develop external functional architecture
   - .3 Develop software internal architecture
   - .4 Assess architected solution vs. requirements
   - .5 Revise architecture and/or renegotiate requirements
   - .6 Document architecture and/or changed requirements
3. DEVELOP EXTERNAL FUNCTIONAL SPECIFICATION
   - .1 Define functional specification standards and conventions
   - .2 Formalize external environment and interface specifications
   - .3 Refine, formalize, and document the architected external operational view of the software
   - .4 Define functional acceptance tests
   - .5 Verify compliance of the external view with requirements
4. PRODUCE AND DELIVER SOFTWARE ITEMS
   - .1 Define programming, test and verification, QA, and documentation standards and conventions
   - .2 Formalize internal environment and interface specifications
   - .3 Obtain support tools
   - .4 Refine and formalize the internal design
   - .5 Define testing specifications to demonstrate required performance
   - .6 Define QA specifications
   - .7 Code and check the program
   - .8 Demonstrate acceptability and deliver software
5. PREPARE FOR SOFTWARE SUSTAINING AND OPERATIONS
   - .1 Train cognizant sustaining and maintenance personnel
   - .2 Train cognizant operations personnel
   - .3 Deliver sustaining tools and materials
   - .4 Deliver all software and data deliverables to operations
   - .5 Install the software and data into its operational environment
   - .6 Prepare consulting agreement between implementation and operations
6. PERFORM PROJECT MANAGEMENT FUNCTIONS
   - .1 Define project goals and objectives
   - .2 Scope and plan the project
   - .3 Administrate the implementation
   - .4 Evaluate performance a[illegible] [illegible]roduct
   - .5 Terminate the project

References

1. Avots, I., "Why Does Project Management Fail?" California Management Review, Fall 1969, Vol XII, No. 1, Pages 77-82.

2. Tausworthe, Robert C., Standardized Development of Computer Software, Prentice-Hall, Englewood Cliffs, N.J., 1977.

3. Hajek, V. G., Management of Engineering Projects, McGraw-Hill Publishing Co, New York, N.Y. 1977.

4. DoD and NASA Guide, PERT/COST, Office of The Secretary of Defense and NASA, Washington, D.C., June, 1962.

5. Tausworthe, Robert C. "Stochastic Models for Software Project Management", Deep Space Network Progress Report no. 42-37, Jet Propulsion Laboratory, Pasadena, CA., February. 1977, pages 118-126.

6. McKenzie, M., and Irvine, A. P., "Evaluation of The DSN Software Methodology", Deep Space Network Progress Report no. 42-46, Jet Propulsion Laboratory, Pasadena, CA., August, 1978.

7. Lehman, M. M., et. al., Software Phenomenology, working papers of The Software Life Cycle Management Workshop, U.S. Army Institute for Research in Management Information and Computer Science, Atlanta, GA., August, 1977.

OPERATION OF THE SOFTWARE ENGINEERING LABORATORY*

Victor R. Basili and Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park, Maryland 20742

Abstract

The paper discusses the current status of the Software Engineering Laboratory. Data is being collected and processed during the development of several NASA/Goddard Space Flight Center ground support projects. The data is used to evaluate software development disciplines and various models and measures of the software development process. Emphasis is placed upon models of resource estimation, the analysis of error and change data, and program complexity measures.

- - - - -

The Software Engineering Laboratory is a research project between NASA/Goddard Space Flight Center and the Department of Computer Science of the University of Maryland. Ground support software, in the six to twelve man-year range, developed for the Systems Development Section of NASA, is studied in detail for determining the dynamics of software development and the effects of various features and methodologies on this development [Basili and Zelkowitz 77]. Most data is collected in a set of reporting forms that are either filled out periodically by all project personnel (e.g., a weekly Component Status report) or whenever certain events occur (e.g., a Change Report Form when an error is corrected). This report describes the activities of the laboratory for the last twelve months.

The initial goal of the Software Engineering Laboratory was the collection of valid data and the entering of this data into a computerized data base. During the last twelve months, this process has been implemented and the analysis of the data has begun. This report will be divided into four sections briefly outlining each of the major activities undertaken by the laboratory: (1) Data Collection Activities, (2) Resource Estimation, (3) Error Analysis, and (4) Program Complexity.

Data Collection Activities

The first task of the laboratory was to implement a data base that accurately reflected software development. The INGRES data base system operating under the UNIX operating system on a PDP 11/45 computer at the University of Maryland was chosen as the basic data base system [Stonebraker 76]. This activity resulted in the following steps:

A generalized table-driven program was implemented that converted the raw typed-in forms to a format acceptable to INGRES. However, it soon became apparent that the major problems were not program oriented, but were in the human communication necessary to carry out this activity.

Forms were frequently filled out containing names not yet recognized by the data base. Other fields were sometimes missing or unclear. Constant interaction between the University personnel and the programmers filling out the forms became necessary in order to solve this problem.

Thus, the first change in procedure was to rewrite the data validation program for the PDP 11/70 at NASA. Forms are turned in to a single individual assigned to the Laboratory. The form is scanned manually and any errors are brought to the attention of the programmers. The validation program finds additional errors that can be quickly corrected. Correct forms are written to tape for transmittal to the University.

This activity led to a second task--a revision of the forms. We observed that the programmers preferred a "checklist" format rather than a set of "fill in the blanks," even if more checks were needed than blanks. Many of the early forms were studied for typical responses and the forms were modified appropriately. In addition, some seemingly useful information, but based upon data that was generally not being given by the programmers, has been deleted in order to lessen the apparent overhead perceived by the programmers participating in the laboratory.

Another activity now under way is the movement of the data base to the PDP 11/70 at NASA. Due to the smaller size of the PDP 11/45 at the University and the relative inefficiency of INGRES for large-scale applications, operation of the University setup is starting to become cumbersome. The PDP 11/70 should eliminate that problem.

Summarizing the activities of the past year, several schemes were developed and we now have evolved a semi-automatic process for entering data into a data base:

*Research supported in part by grant NSG-5123 from NASA/Goddard Space Flight Center to the University of Maryland.

1. Forms are turned in and manually scanned for errors.
2. The forms are entered into a validation program at NASA. If errors are present, the form is returned for corrections. If correct, it is written to tape.
3. The tape of correct forms is brought to the University for data base entry.

(By January 1979, it is expected that the corrected tape will also be entered into a data base on NASA's PDP 11. At that time the decision will be made as to whether to keep the University data base or to interface with NASA's.)

Resource Estimation

One early research activity was the investigation of resource utilization. The Rayleigh curve has been studied for larger projects and the applicability of this theory in the smaller NASA environment was investigated.

Cumulative costs for large-scale software development has been shown to approximate the curve $K(1 - e^{-at^2})$ where K is the total project cost and t is the elapsed time since project initialization [Putnam 76]. This is usually represented in its differential form called a Rayleigh curve: $(2Kate^{-at^2})$, and represents the rate of consuming resources. This curve looks somewhat like a normal distribution with a more extended tail (see Figure 2).

In our NASA environment, from the general project summary form, these numbers are obtained:

1. K_e, total estimated cost of the project in hours of effort. Counting overhead items, like typing support and librarians, total costs (K) are usually 112% of K_e.
2. Y_d, the maximal effort per week. From this, constant a can be developed, $a = (1/2y_d^2)$.
3. T_a, the estimated date of acceptance testing. In NASA's environment this usually occurs after 88% of total expenses are consumed.

Since the Rayleigh curve has two parameters (K and a) and the general project summary gives three (ie, y_d, and t_a), the applicability of the Rayleigh curve to this environment can be checked by using two of these estimates to predict the third.

Figure 1 represents this analysis for two projects. Figure 1.A presents the estimated data from the general project summary. In Figure 1.B, t_a was estimated from K_e and y_d and y_d was estimated from K_e and t_a. Finally, Figure 1.C presents the actual data.

Figure 2 plots some of this for these two projects. While Figure 1 shows that K_e and y_d are accurate predictors of t_a (e.g., an estimate of 60 weeks for project A, only a two-week error from the actual 62 weeks, and a much better estimate than the initial estimate of 46 weeks), the plots of this curve differ from actual resource consumption. The conclusion seems to be that the Rayleigh curve is only a crude approximation to reasonable consumption. (See [Basili and Zelkowitz 78a] for more details.)

In order to test this further, several other curves were correlated with the actual data (parabola, trapezoid and straight line) [Mapp 78]. All had as good correlations to the data as the Rayleigh curve. Thus, the Rayleigh curve was no better, and in many cases worse, than other estimates.

In addition, Norden's original assumptions involve a linear growth in the rate of understanding a project [Norden 70]. In reality, this learning curve slows as personnel become familiar with a project. Based upon this assumption, [Parr 78] has developed a curve based upon the hyperbolic secant that may be more applicable in the NASA environment. This and other theories related to the Rayleigh curve are now being studied.

The evaluation performed in [Basili and Zelkowitz 78a] has led to a set of procedures that can be used to monitor project development in a production environment. While the full set of seven reporting forms may prove to be too much overhead, a set of procedures using only three forms can be used to monitor project progress with reasonable accuracy [Basili and Zelkowitz 78b]: the General Project Summary, submitted at each project milestone; the Resource Summary, giving hours worked by all project personnel by week; and the Change Report Form giving all changes to the system.

Error Analysis

The principal motivations for studying errors and changes have been to discover the effects of various factors on the number and kinds of errors made in system developments, and to find ways to evaluate proposed software development methodologies.

To study this, a number of tasks have been performed. First, to assure that all of the forms have been filled out in a consistent manner, a glossary of terms has been defined and made available to the participants of the monitored software development process. Second, a set of questions of interest were defined which were used to moti[illegible]te both the form content and organize the kinds of data required in the form of interviews with the participants. Questions of interest include the following:

What are good ways of characterizing error-proneness of software development? Measures such as the total number of errors, errors per line of code, errors per man hour, errors per component type where type refers to the kind of sub-application or level of complexity, number of fixes per project phase per component are being considered. We are also looking at relationships between the various types of error classification.

What are the major sources of errors? One possible characterization is by analyzing whether errors are traceable back to requirements, specification, interface definition or intra-component design, or clerical activities or the hardware environment.

What are appropriate ways of measuring ease of software change? Data is being collected on effort per change in terms of time, the number of fixes required for the change, and the number of errors generated by the change.

What is the effect of continual change on a software product? Data is gathered on the cost of change as a function of time and cumulative changes.

What type of changes cause most of the errors? This may be very environment dependent or it may give some insights into improved organizations and methodologies for software development.

What is the effect of personnel organization on errors? Data is being collected on correctness as measured by errors per number of people working on a piece of software. Again, this should shed some insights on the way to organize tasks within a given environment.

What types of changes predominate during software development? Knowing this should aid in designing software to anticipate the possible changes.

What are the most prevalent error detection and correction techniques? Knowing what is used most often and what works and at what cost will help in determining what should be used for what classes of errors in what environment.

What is the effect of various constraints, such as time and memory on error distributions? Understanding this will permit better evaluation of the tradeoffs in software management.

These are but some of the types of questions the error analysis phases of the Software Engineering Laboratory is studying. The data for most of these questions is gathered from the Change Report Form, with additional information from the other forms and follow-up interviews to validate the accuracy of the information and gather additional data not easily collected in a form format.

Based upon the above questions, several "first order metrics" have been defined and software has been developed to gather information from the data base. Data is being gathered at a slow pace partly because of the current backlog of Change Report Forms which have not yet been entered into the data base, and partly because of the refinement of the form as mentioned in the section on Data Collection Activities. Early analyses on a couple of projects, however, do indicate that the distribution of errors during development appears to approximate the Rayleigh curve as found by [Schick and Wolverton 78].

Continued effort will deal with the gathering of information to answer the basic questions of interest, further development of new questions of interest, and possible "second order metrics" based on the intuition gathered from the current studies.

Program Complexity

There is much interest in measures of complexity of the software product, the valid aspects of the product that effect human understanding. There is an interest in quantitatively measuring these aspects so that characteristics of programs that make them more or less error prone, harder to modify, or more difficult to develop can be better understood and recognized. Measures proposed in the literature may even be used to characterize differences in the development process.

Work has been done at the University of Maryland to analyze and compare the development of software in an experimental environment to determine the effects of development methodologies [Basili and Reiter 78]. The experiment involved the use of three different types of development: Single individuals using ad hoc techniques, groups of three using ad hoc techniques, and groups of three using a structured programming methodology. Results have shown that there is some distinction in the product using very rough measures of the program characteristics, such as number of *if* statements, number of globals, etc. Based on this study, the organized group lies somewhere between the ad hoc group and the single individual. However, with regard to process measures, the organized group has shown less computer runs in all phases of development and less errors (using a measure of errors called program changes which is algorithmically computable based on different versions of the software product [Dunsmore 78]).

It is planned to implement the promising measures from this research on programs from the NASA environment. Versions of the systems developed at NASA have been saved and will be compared for program changes and checked against the result from the Error Report in the Change Report Forms. Further work is being done in automating and comparing various complexity measures. These include several of our own measures (prime program hierarchy, data bindings, etc.) as well as some of the measures that have appeared in the literature [Halstead 77; McCabe 76].

References

[Basili and Zelkowitz 78a] Basili, V. and M. Zelkowitz, Analyzing Medium-Scale Software Development, Third International Conference on Software Engineering, Atlanta, Georgia, May 1978, pp. 116-123.

[Basili and Zelkowitz 78b] Basili, V. and M. Zelkowitz, Measuring Software Development Characteristics in the Local Environment, Journal of Computers and Structures, 1978, 5 pp. (to appear).

[Basili and Zelkowitz 77] Basili, V. and M. Zelkowitz, The Software Engineering Laboratory: Objectives, ACM SIGCPR Annual Conference, Washington, D. C., August 1977, pp. 256-269.

[Basili and Reiter 78] Basili, V. and Reiter, R. An Experimental Comparison of Software Development Approaches, University of Maryland, Computer Science Technical Report TR-688, August 1978.

[Dunsmore 77] Dunsmore, H. E. and Gannon, J. D., Experimental Investigation of Programming Complexity, Proceedings of the 16th Annual Technical Symposium: Systems and Software Washington, D. C. (June 77) pp. 117-125.

[Halstead 77] Halstead, M., Elements of Software Science, Elsevier Computer Science Library 77.

[McCabe 76] McCabe, Thomas J., A Complexity Measure, Transactions on Software Engineering, Dec. 76, Vol. SE-2, No. 4, pp. 308-320.

[Mapp] Mapp, T., Applicability of the Rayleigh Curve to the SEL Environment, University of Maryland, Department of Computer Science, Scholarly Paper, May 1978.

[Norden 70] Norden, P., Use Tools for Project Management, Management of Production, M. K. Starr (ed), Penguin Books, Baltimore, Maryland, 1970, pp. 71-101.

[Parr 78] Parr, F., An Alternative to the Rayleigh Curve Model for Software Development Effort (submitted for publication).

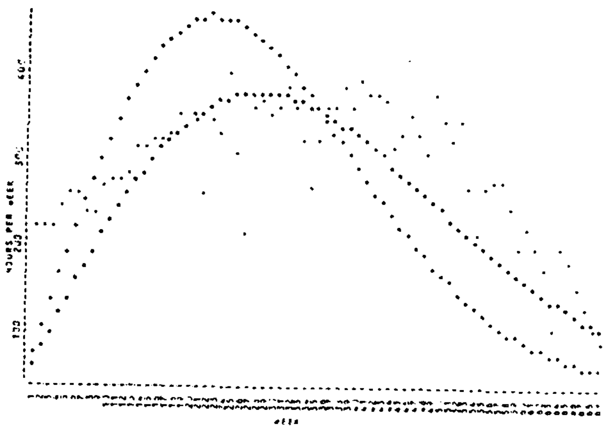
[Putnam 76] Putnam, L., A Macro-estimating Methodology for Software Development, IEEE Computer Society Compcon, Washington, D. C., September 1976, pp. 138-143.

[Schick and Wolverton 78] Schick, George J. and Wolverton, Ray W., An Analysis of Competing Software Reliability Models, IEEE Transactions on Software Engineering, Vol. SE-4, No. 2, March 1978, pp. 104-120.

[Stonebraker 76] Stonebraker, M., E. Wong and P. Kreps, The Design and Implementation of INGRES, ACM Transactions on Data Base Systems 1, No. 3, 1976, pp. 189-222.

	PROJECT A	PROJECT B
A. Initial Estimates from General Project Summary		
Ka, Resources needed (hrs)	14,213	12,997
Ta, Time to completion (wks)	46	41
Yd, Maximum resources/wk (hrs)	350	320
B. Completion Estimates Using Rayleigh Curve		
K, Resources needed (hrs)	16,151	14,770
Est. Yd with Ta fixed (hrs)	440	456
Est. Ta with Yd fixed (hrs)	58	58
C. Actual Project Data		
K, Resources needed (hrs)	17,741	16,543
Yd, Maximum resources (hrs)	371	462
Ta, Completion time (wks)	62	54
Ta, Estimated using actual values of K and Yd (wks)	60	43

Figure 1: Estimating Ta and Yd from General Project Summary Data



* - Estimating curve with Yd (maximum resources) fixed
+ - Estimating curve with Ta (completion date) fixed
. - Actual data

Figure 2. Estimated resource expenditures curve

LIFE CYCLE MANAGEMENT MEASUREMENT
METRICS-MEASURES & EMPIRICAL STUDIES

SOME DISTINCTIONS BETWEEN THE PSYCHOLOGICAL AND COMPUTATIONAL COMPLEXITY OF SOFTWARE

Bill Curtis, Sylvia B. Sheppard, M.A. Borst, Phil Milliman, and Tom Love

Information Systems Programs
General Electric Company
Arlington, Virginia

ABSTRACT

Three software complexity metrics (number of statements, McCabe's v(G), and Halstead's E) were compared to performance on two software maintenance tasks. In an experiment on understanding, length and v(G) correlated with the percent of statements correctly recalled. In an experiment on modification, most significant correlations were obtained with metrics computed on modified rather than unmodified code. All three metrics correlated with time to complete the modification, while only length and v(G) correlated with the accuracy of the modification. Relationships in both experiments occurred in unstructured rather than structured code, and in the second experiment where no comments appeared in the code. The metrics were also most predictive of performance for inexperienced programmers. Thus, these metrics appeared to assess psychological complexity only where programming practices did not provide assistance in understanding the code. In both experiments all three metrics were highly intercorrelated.

INTRODUCTION

In 1972, Halstead first published his theory of software physics (renamed software science) stating that algorithms have measurable characteristics analogous to physical laws. According to Halstead (1972a, 1972b, 1975, 1977) the amount of effort required to generate a program can be calculated from simple counts of distinct operators and operands and the total frequencies of operators and operands. From these four quantities Halstead calculates the number of mental comparisons required to generate a program. Halstead's metrics attempt to represent the psychological complexity of software. Correlations often greater than .90 (Fitzsimmons & Love, 1978) have been reported between Halstead's metrics and such measures of programmer performance as the number of bugs in a program (Cornell & Halstead, 1976; Fitzsimmons, 1978; Funami & Halstead, 1975), programming time (Gordon & Halstead, 1975; Halstead, 1976), and the quality of programs (Bulut & Halstead, 1974; Elshoff, 1976; Halstead, 1973).

More recently McCabe (1976) developed a definition of complexity based on the decision structure of a program. McCabe's complexity metric is the classical graph theory cyclomatic number which represents the number of regions in a graph, or in the current usage, the number of linearly independent control paths comprising a program. Simply stated, McCabe's metric counts the number of basic control paths. When combined these paths will generate every possible path through the program. Since McCabe attempted to relate his metric to the difficulty of testing a program, it was presented as a measure of computational complexity. Nevertheless, the number of basic control paths indexed by McCabe's metric may also be an important aspect of psychological complexity, since additional control paths could make a program more difficult to understand.

Since the reasons for assessing them are completely different, psychological and computational complexity should be clearly distinguished when interpreting software complexity metrics. Computational complexity refers to characteristics of algorithms or programs which make their proof of correctness difficult, lengthy, or impossible (Rabin, 1977). For example, as the number of distinct control paths through a program increases, the computational complexity also increases. Psychological complexity refers to characteristics of software which make it difficult to understand and work with. Thus, computational complexity assesses the difficulty of verifying an algorithm's correctness, while psychological complexity assesses human performance on programming tasks. No simple relationship between computational and psychological complexity is expected. For example, a program with many control paths may not be psychologically complex, since any regularity to the program's branching process may simplify its understanding.

The research reported here was designed to investigate factors influencing two tasks in software maintenance: understanding an existing program and implementing modifications to it. These factors included structured programming techniques, cognitive programming aids, and program complexity. While the first two factors were manipulated experimentally, no systematic attempt was made to manipulate program complexity. This paper reports data from two experiments which investigated how several complexity metrics were related to the understanding (Experiment 1) and modifying (Experiment 2) of computer programs.

EXPERIMENT 1

Method

Participants. Thirty-six programmers were tested in five different General Electric locations. The participants had working knowledge of FORTRAN and averaged 6.8 years of professional programming experience (SD = 5.8). Most participants came from an engineering background, while several were experienced in statistical or non-numerical software.

Procedure. Each participant was presented a packet of materials with written instructions on the experimental tasks. As a preliminary exercise, all participants were presented the same short FORTRAN program and a brief description of its purpose. They studied this program for 10 minutes and were then given 5 minutes to reconstruct a functional equivalent from memory. This introductory program diminished learning effects prior to the experimental phase.

Following the initial exercise, participants were presented sequentially with three separate programs comprising their experimental tasks. They were allowed 25 minutes to study each program and could make notes or draw flowcharts. At the end of the study period, the original program and all scrap paper were collected. Each participant was then given 20 minutes to reconstruct a functionally equivalent program from memory on a blank sheet of paper, but was not required to reproduce the comment section. A break of 15 minutes occurred before the last program was presented.

Programs. Three general classes of programs were used: engineering, statistical, and non-numerical. Three programs were employed from each class with lengths varying from 36 to 57 statements. These nine programs were selected from among many solicited from programmers at several locations and were considered representative of programs actually encountered by practicing programmers. All experimental programs were executed using appropriate test data.

Complexity of control flow. Three control flow structures were defined for each program. Structured control flow adhered very strictly to the tenets of structured programming (Dijkstra, 1972). When the rules for structured programming are applied rigorously,awkward constructions may occur in standard FORTRAN, such as DO loops with dummy indices (Tenny, 1974). In a second version these awkward constructions were largely eliminated with a more naturally structured control flow. These conventions included multiple returns, backward exits from DO loops, and judiciously used backward GO TO's. In the unstructured version of each program the control flow was not straightforward. Expanded DO loops, arithmetic IF's, and unrestricted use of GO TO's were allowed.

Variable name mnemonicity. Three levels of mnemonicity for variable names were manipulated independently of program structure. The least mnemonic names consisted of one or two alphanumeric characters, while the most mnemonic names were chosen from names suggested by a group of programmers not participating in the study.

Experimental design. In order to control for individual differences in performance, a within-subject, 3^4 fractional factorial design was employed (Hahn & Shapiro, 1966; Kirk, 1968). Three types of control flow were defined for each of nine programs, and each version was presented in three levels of variable mnemonicity for a total of 81 experimental programs.

Halstead's E. Halstead's effort metric (E) was computed precisely from a program (based on Ottenstein, 1976) whose input was the source code listings of the 27 programs representing nine distinct programs at each of three levels of structure. The computational formula was:

$$\underline{E} = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}$$

where, η_1 = number of unique operators

η_2 = number of unique operands

N_1 = total frequency of operators

N_2 = total frequency of operands

McCabe's v(G). McCabe's metric is the classical graph-theory cyclomatic number defined as:

$$\underline{v(G)} = \#\text{ edges} - \#\text{ nodes} + 2\ (\#\text{ connected components}).$$

McCabe presents two simpler methods of calculating v(G). For structured programs v(G) equals the number of predicate nodes plus 1. Values of v(G) can also be computed from a planar graph of the control flow by counting the number of regions.

Length. The length of the program was computed as the total number of FORTRAN statements excluding comments.

Dependent variable. The criterion for scoring programs was the functional correctness of each separately reconstructed statement. Variable names and statement numbers which differed from those in the original program were counted as correct when used consistently. Control structures could be different from the original program so long as the statement performed the same function. The score on each experimental task was the percent of statements correctly recalled.

Results

Experimental manipulations. A complete report of the experimental results is presented in Sheppard, Borst, Curtis, and Love (in press). Briefly, a mean of 50% of the statements were correctly recalled across all programs and experimental conditions. Substantial differences in performance were observed across the nine programs. Performance on naturally structured programs was superior to that on unstructured programs. Differences in the mnemonicity of the variable names did not affect performance.

Software complexity metrics. Since different levels of variable mnemonicity neither affected performance, nor caused any change in the value of the complexity metrics for a particular program, the data reported in this section were aggregated over the three levels of variable mnemonicity on each of the nine programs at each of the three levels of structure. Thus, each of the 27 data points represented the average of at least three performance scores.

Halstead's E and McCabe's v(G) were highly correlated ($r = .84$, $p \leq .001$), while length displayed only moderate relationships with these two metrics. The correlations between performance and the complexity metrics were all negative, indicating that fewer lines were recalled as the level of complexity represented by these three metrics increased. Performance was moderately related to length ($r = -.53$, $p \leq .01$) and McCabe's v(G) ($r = -.35$, $p \leq .05$), but not to Halstead's E.

The complexity of the control flow moderated the relationship between performance and the complexity metrics. That is, while insignificant correlations were observed when the control flow was structured or naturally structured, this was not the case for unstructured code. Correlations with performance of .55 ($p \leq .001$) and .45 ($p \leq .01$) for v(G) and E were observed on unstructured programs.

A similar moderating effect was observed for a programmer's extent of professional experience. For programmers with three or less years of professional experience, correlations of -.47 ($p \leq .001$) for McCabe's v(G) and -.35 ($p \leq .05$) for Halstead's E were observed. Insignificant correlations were observed for programmers with more than three years experience.

EXPERIMENT 2

Method

Participants. As in the previous experiment, the sample for this experiment consisted of 36 professional programmers from three General Electric locations. The participants averaged 5.9 years of professional programming experience (SD = 4.1), had a working knowledge of FORTRAN, and none had participated in the previous experiment.

Procedure. Generally, the procedures employed in this experiment were identical to those used in Experiment 1. In a preliminary exercise, all participants were asked to modify the same short FORTRAN program. Following this initial exercise, participants were presented in turn with the three programs comprising their experimental tasks. One modification was requested for each program and was described on a sheet accompanying the program listing. Participants were allowed to work at their own pace, taking as much time as needed to implement the modification.

Experimental design. In order to control for individual differences in performance, a within-subject, 3^4 factorial design was employed. Three of the nine programs from Experiment 1 were used. Three types of control flow were defined for each of the three programs using the same approach described previously. Each of these nine versions was presented with one of three types of commenting. Modifications at three different levels of difficulty were developed for each program generating a total of 81 experimental programs. Participants were randomly assigned into the experimental design.

Comments. Three levels of commenting were tested in this experiment: global, in-line, and none. Global comments provided an overview of the function of the program and identified the primary variables. In-line comments were interspersed throughout the program and described the specific functions of small sections of code.

Modifications. Three types of modifications were selected for each program as typical changes a programmer might be expected to implement. The level of difficulty for seven of the nine modifications increased as more lines had to be added to the original code, and the hardest modification for each program required the most additional lines.

Dependent variables. The dependent variables were the correctness of the modification and the time taken by the participant to complete the task. The individual steps necessary for the correct implementation of each requested modification were delineated in advance and assigned equal weights. That is, prototypes of each program with each modification correctly implemented were established as the criteria against which participants' work would be compared. A percentage score representing the functional correctness of each modification was computed by comparing a participant's changes with the prototype version. All values for the complexity metrics computed on the modified programs were computed on the prototypes with correct implementations rather than from code generated by the participants. The time to implement a modification was timed to the nearest minute using an electronic timer.

Results

Experimental manipulations. A complete report of the results for the manipulations in this experiment appears in Sheppard, Borst, Curtis, and Love (in press). Briefly, across all experimental conditions an average accuracy score of 62% was received on the modifications (SD = 31%). The 108 accuracy scores ranged from five scores of 0% to 24 scores of 100% and were negatively skewed. The average time to complete the modification was 17.9 minutes (SD = 11.4), ranging from 2 to 59 minutes with a positive skew. Accuracy and time were uncorrelated.

On two of the three programs studied, the accuracy of the modification was found to be modestly affected by both the difficulty of the modification and the use of structured programming techniques.

More accurate modifications were made to strictly structured rather than unstructured programs. The more difficult modifications took longer to implement, although structured programming techniques had no effect on time to completion. Neither time nor accuracy was affected by the type of commenting.

Software complexity metrics. Correlations among Halstead's and McCabe's metrics and length were quite high on both the original and modified programs ($.88 \leq r \leq .97$, $p \leq .001$). Correlations between the three complexity metrics and the two dependent variables were larger in the aggregated than in the unaggregated data. Most significant correlations with performance were observed for metrics computed on the modified programs. All three metrics were moderately correlated with time to complete the modification ($.38 \leq r \leq .46$, $p \leq .05$), while only length and McCabe's $v(G)$ were significantly related to accuracy ($-.34 \leq r \leq -.36$, $p \leq .05$).

Similar to results in Experiment 1, the relationship between time to completion and Halstead's E was moderated by the complexity of the control flow. The correlations for Halstead's E with performance went from .08 in the structured code, to .28 ($p \leq .05$) in naturally structured code, to .38 ($p \leq .05$) in unstructured code. No such moderating effects were observed for McCabe's $v(G)$.

Correlations between the complexity metrics and performance measures were also moderated by the type of commenting. Significant correlations on modified programs were observed when no comments were included in the program for both accuracy ($-.34 \leq r \leq -.35$, $p \leq .05$) and time ($.44 \leq r \leq .47$, $p \leq .01$).

The amount of professional programming experience profoundly moderated the relationships observed between the complexity metrics and time to completion, although no such effect was observed for accuracy. Significant correlations ($.52 \leq r \leq .55$, $p < .001$) were observed for programmers with three or less years of professional experience, while no correlations above .20 were observed for programmers with more than three years experience.

CONCLUSIONS

The two experiments comprising this study produced empirical evidence that software complexity metrics were related to the difficulty programmers experienced in understanding and modifying programs. Deeper analysis indicated, however, that the Halstead and McCabe metrics predicted programmers' performance only on certain programs. Programs on which significant prediction was observed were characterized by the absence of programming practices such as structured coding or commenting which provide assistance to a programmer attempting to understand the code. These complexity metrics were more predictive of the performance of less experienced programmers.

Assessment of the psychological complexity of software appears to require more than a simple count of operators and operands or basic control paths. Many programs have characteristics unassessed by these metrics which may heavily influence psychological complexity. For instance, the use of structured coding techniques or comments reduces the cognitive load on a programmer in ways unassessed by the complexity metrics. Further, complexity metrics may not be capturing the most important constructs for predicting the performance of experienced programmers who may either be conceptualizing programs at a level other than that of operators, operands, and basic control paths, or who can fit the program into a schema similar to one they have had previous experience with.

Even though moderating effects were observed in these data, stronger relationships with performance may have been masked by the effects of differences between individuals and programs which were enhanced by limitations in the economical multifactor designs employed. Uniformity in the sizes of programs studied may also have limited these results. The range of values assumed by complexity metrics computed on these programs may have been insufficient for correlational tests to detect the strong relationships reported in other verifications of these theories (Edwards, 1976). Studies reporting higher correlations for Halstead's E usually involved a broader range of program sizes (Fitzsimmons & Love, 1978, Halstead, 1977).

The number of statements in the code proved to be as good a predictor of performance on the experimental tasks as the metrics developed by Halstead and McCabe. These results did not concur with those of Gordon (1977) who, in reanalyzing Love's (1977) data on programmer understanding, found Halstead's metric better than the number of statements for predicting the percentage of statements recalled. Correlations among the metrics suggested substantial overlap in the constructs they quantified on the modular-sized programs studied. Had these correlations been smaller, the predictive worth of the metrics could have been better compared. The number of statements may not be as highly correlated with the Halstead and McCabe metrics for programs of substantially greater length, for other types of applications, or for programs written in languages other than FORTRAN.

A characteristic distinguishing psychological from computational complexity is that the psychological complexity of software involves an interaction between program characteristics and individual differences, such as programming experience. Chrysler (1978) demonstrated the value of these experiential variables in predicting the time to complete a programming task. Individual differences should not be overlooked in predicting human performance, especially when the performance ratio comparing good to bad programmers has been reported as high as 28 to 1 (Grant & Sackman, 1967).

The complexity metrics provided some sources of information about program differences, but there were other factors within the programs unassessed by these metrics which may have influenced psychological complexity. Neither Halstead's nor McCabe's metrics consider the level of nesting within various constructions. The complexity of three DO loops in succession was rated identically to three nested DO loops, although nesting may influence complexity. If the ability of complexity metrics to predict human performance on programming tasks is to be improved, then metrics must be designed which measure phenomena related by psychological principles to memory, information processing, and problem solving. Thus, while the number of control paths may be critical in computational complexity, variations in the arrangement and connections among these control paths may exert profound influence on the difficulty of understanding the functioning of the program. Future work in the area of psychological complexity should identify a set of cognitive psychological principles relevant to programming tasks. Metrics could then be developed which assess the qualities of software which are most closely related to these principles. Such an exercise might not only lead to improved metrics for assessing psychological complexity, but may also identify some programming practices which could lead to simplified, more easily maintained software.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the assistance of Ann Fitzsimmons in implementing this research and of Dr. Gerald Hahn in developing the experimental design. Careful reviews of this report by Dr. John O'Hare and Thomas McCabe have resulted in substantial improvements. The support and encouragement of Lawrence Putnam and Gerald Dwyer has also been greatly appreciated.

This study was supported by the Office of Naval Research, Engineering Psychology Programs (Contract #N00014-77-C-0158). The views expressed in this paper, however, are not necessarily those of the Office of Naval Research or the Department of Defense.

Expanded reports of each experiment can be obtained by writing the senior author at: GE/ISP, Suite 200; 1755 Jefferson Davis Hwy.; Arlington, VA 22202.

REFERENCES

Bulut, N., & Halstead, M.H. Impurities found in algorithm implementation (Tech. Rep. CSD-TR-111). West Lafayette, IN: Purdue University, Computer Science Department, 1974.

Chrysler, E. Some basic determinants of computer programming productivity. Communications of the ACM, 1978, 21, 471-483.

Cornell, L., & Halstead, M.H. Predicting the number of bugs expected in a program module (CSD-TR-205). West Lafayette, IN: Purdue University, Computer Science Department, 1976.

Dijkstra, E.W. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, & C.A.R. Hoare (Eds.) Structured programming. New York: Academic Press, 1972.

Edwards, A.L. An introduction to linear regression and correlation. San Francisco: Freeman, 1976.

Elshoff, J.L. Measuring commercial PL/1 programs using Hlastead's criteria. SIGPLAN Notices, 1976, 11, 38-46.

Fitzsimmons, A.B. Relating the presence of software errors to the theory of software science. In A.I. Wasserman & R.H. Sprague (Eds.) Proceedings of the eleventh Hawaii international conference of systems sciences (Vol. 1). San Francisco: Western Periodicals, 1978.

Fitzsimmons, A.B. & Love, L.T. A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.

Funami, Y., & Halstead, M.H. A software physics analysis of Akiyama's debugging data (Tech. Rep. CSD-TR-144). West Lafayette, IN: Purdue University, Computer Science Department, May 1975.

Gordon, R.D. A measure of mental effort related to program clarity. Unpublished doctoral dissertation. Purdue University, 1977.

Grant, E. E., & Sackman, H. An exploratory investigation of programmer performance under on-line and off-line conditions. IEEE Transactions on Human Factors in Electronics, 1967, HFE-8, 33-48.

Hahn, G. J., & Shapiro, S. S. A catalogue and computer program for the design and analysis of orthogonal symmetric and asymmetric fractional factorial experiments (Tech. Rep. 66-C-165). Schenectady, NY: General Electric, May 1966.

Halstead, M. H. Natural laws controlling algorithm structure. SIGPLAN Notices, 1972, 7, 2. (a)

Halstead, M. H. A theoretical relationship between mental work and machine language programming. (Tech. Rep. CSD-TR-67). West Lafayette, IN: Purdue University, Computer Science Department, May 1972. (b)

Halstead, M. H. An experimental determination of the "purity" of a trivial algorithm. (Tech. Rep. CSD-TR-73). West Lafayette, IN: Purdue University, Computer Science Department, 1973.

Halstead, M. H. Software physics: Basic principles. (Tech. Rep. RJ-1582). Yorktown Heights, NY: IBM, 1975.

Halstead, M. H. Using the methodology of natural science to understand software (Tech. Rep. CSD-TR-190). West Lafayette, IN: Purdue University, Computer Science Department, 1976.

Halstead, M. H. Elements of software science. New York: Elsevier North-Holland, 1977.

Kirk, R. E. Experimental design procedures for the behavioral sciences. San Francisco: Freeman, 1968.

Love, L. T. Relating individual differences in computer programming performance to human information processing abilities. Unpublished doctoral dissertation, University of Washington, 1977.

McCabe, T. J. A complexity measure. IEEE Transactions on Software Engineering, 1976, SE-2, 308-320.

Ottenstein, K. J. A program to count operators and operands for ANSI-FORTRAN modules. (Tech. Rep. CSD-TR-196). West Lafayette, IN: Purdue University, Computer Science Department, June 1976.

Rabin, M. O. Complexity of computations. Communications of the ACM, 1977, 20, 625-633.

Sheppard, S. B., Borst, M. A., Curtis, B., & Love, T. Factors influencing the understandability and modifiability of computer programs. Human Factors, in press.

Tenny, T. Structured programming in FORTRAN. Datamation, 1974, 20, 110-115.

Gordon, R.D., & Halstead, M.H. An Experiment comparing FORTRAN programming times with the software physics hypothesis (Teach. Rep. CSD-TR-167). West Lafayette, IN: Purdue Univ., Computer Science Dept., 1975.

A Review of Software Measurement Studies at General Motors Research Laboratories

James L. Elshoff

Computer Science Department
General Motors Research Laboratories
Warren, Michigan 48090

The software measurement studies began at General Motors Research in 1973 with the development of a PL/I scanner. The scanner was written to make one pass over the code to be scanned and to look ahead one token. The scanner was modularized so that various counters could easily be inserted into it to record the occurrence of particular constructs in the PL/I programs to be scanned. The recorded data could then be studied by using a small interactive query system that existed on the same computer as the scanner.

In 1974 a large volume of PL/I code was collected from five representative General Motors data processing centers. A subset of 120 of the collected programs that comprised over 100,000 PL/I source statements were used for a controlled empirical study. The significant data from this study are published [1] with only minor interpretation in order that anyone may study the data from a particular point of view. The data indicate how the PL/I language was being used in the early 1970's. An averge compilable unit for example was made up of a single procedure containing 853 PL/I statements. Control structures for those programs were very complex. Data handling and expressions were found to be relatively simple but voluminous.

The data that had been collected were then analyzed [2] with respect to good programming practices. Most of these practices fall into the class described generically as structured programming. The reason for using these practices lies in common sense and just a few published experiences even though the practices are widely touted. The program units were observed to be too large since an average program of 853 PL/I statements contained 386 identifiers, 1087 constants, 910 expressions, 319 flow of control statements (including 100 GO TO statements), and 50 statement labels. The programs were found to be extremely difficult to read. The programs were determined to be extremely complex with respect to both control flow and data flow. The PL/I programming language was used poorly; however, only a limited subset of PL/I was necessary to perform the computing that was required.

A corporate training program was introduced as a result of the analysis that had been performed. Programmers and analysts were introduced to structured design and structured programming techniques. After some programmers had worked with the new techniques for awhile, a new set of programs that had been developed after the training period were collected and analyzed. The analysis of the new programs [3] showed that their profiles reflected the training to which the programmers had been exposed. Although no specific study was performed to determine the overall effectiveness of the new design and programming techniques with respect to the software life cycle, project case studies indicated reductions of over 25% in development time could be achieved. Also, case studies show that maintenance of a program once it has been committed to production can be reduced by as much as a factor of eight.

One of the major problems that surfaced during the analysis of the first set of programs and again during the training program was that modularization seemed foreign to the analysts and programmers. In fact, modularization was viewed as detrimental in all aspects of the software life cycle due to the perceived overhead of CALL/RETURN mechanisms and the cost of packaging programs in small individually compilable units. A concerted effort was made to break down this barrier against modularization in any form. One form of this work was a model of compilation costs [4] that indicates that compilation costs are miniimized when programs are packaged in relatively small units.

The data that was gathered as part of the program analysis phase of our work has

AD-A104 212   INTERNATIONAL BUSINESS SERVICES INC WASHINGTON DC   F/G 9/2
SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP (2ND) AUGUST 21-22, 197--ETC(U)
AUG 78   V R BASILI, E H ELY   DAAK70-78-D-0030
UNCLASSIFIED   78CH1390-4C   NL

3 OF 3
AD A
104212

END
DATE
FILMED
10-81
DTIC

also been used in some experiments with Halstead's software science [5]. The data show that the relationship between measured length and estimated length of programs holds for programs extending over several magnitudes in length [6]. On the other hand, the conjecture that the second set of analyzed programs would be more consistent with respect to the length relationship than the first set of programs was not upheld.

A better understanding of both software science and GM's PL/I programs resulted from applying the theory of software science to the programs. This work also resulted in a joint effort with Halstead and Gordon which led to two significant extensions of software science [7]. A global level of an algorithm which depends only upon a language and its use was derived. This development led to a predictive measure for estimating the time required to write a program. The global level developed in this work is of interest since it ranked an assembler language, FORTRAN, ALGOL 58, poorly used PL/I, well written PL/I, and English in the same order with respect to the level of algorithm expression that one intuitively ranks them.

Even though software science is based on counting operators and operands used to express an algorithm, the exact classification of a language token as an operator or an operand is not always clear. This classification problem was studied in an experiment which applied different counting methods to a fixed set of PL/I programs [8]. Some properties of the algorithms were found to vary significantly depending on the counting method that was used; other properties remained stable. Although no one counting method was shown to be best, the results indicate the importance of the counting method to the overall measurement of an algorithm. Moreover, the results provide a reminder of how sensitive some of the measurements are and of how careful researchers must be when drawing conclusions from software science measurements.

In a recent study [9], several models of operator distributions are compared with the measured distributions found in some PL/I programs. The theory embodies models of operator distributions that have been proposed by Bayer, Zipf, and Zweben. Two of the three models, one by Zipf and one by Zweben, are shown to correlate highly with the measured distributions. Two variations of the Zipf model are then formulated and tested with good results. Also, the relationship between the method of counting operators and the models is investigated.

The work in measurement and empirical studies has been temporarily suspended during the past year while a new PL/I scanner has been implemented. The new scanner builds an abstract tree representation of a program which will permit much more extensive measurement capability than has been available in the past. It is hoped that the next set of programs that are collected for study will have some historical data of their development available to provide a basis for comparison as we continue our program analysis studies.

References

1. --, "A numerical profile of commercial PL/I programs," Software-Practice and Experience, Vol. 6, No. 4, Oct.-Nov. 1976, pp. 505-526.

2. --, "An analysis of some commercial PL/I programs," IEEE Trans. Software Engr., Vol. SE-2, June 1976, pp.113-120.

3. --, "The influence of structured programming on PL/I program profiles," IEEE Trans. Software Engr., Vol. SE-3, Sept. 1977, pp.364-368.

4. --, "On optimal module size with respect to compilation cost," COMPSAC '77 Proceedings, IEEE Computer Society, IEEE Catalog No. 77CH1291-4C, Chicago, Illinois, Nov. 1977, pp. 547-553.

5. Halstead, M. H., <u>Elements of software science</u>, Elsevier North-Holland, Elsevier Computer Science Library, New York, New York, 1977, 127 p.

6. --, "Measuring commercial PL/I programs using Halstead's criteria," SIGPLAN Notices, Vol. 11, No. 5, May 1976, pp. 38-46.

7. Halstead, M. H., --, and Gordon, R. D., "On software physics and GM's PL/I programs," GMR-2175, General Motors Research Laboratories, Warren, Michigan 48090, 26 p.

8. --, "An investigation into the effects of the counting method used on software physics measurements," SIGPLAN Notices, Vol. 13, No. 2, Feb. 1978, pp. 30-45.

9. --, "A study of the structural composition of PL/I programs," SIGPLAN Notices, Vol. 13, No. 6, June 1978, pp. 2937.

SOFTWARE SCIENCE -- A PROGRESS REPORT

M. H. Halstead

Computer Science Department, Purdue University
Lafayette, Indiana 47907

Introduction

At the Airlie House Workshop on Software Life Cycle Management a year ago, a few of us were just becoming aware of the possibility of applying the experimental and theoretical results of software science to the real world problems of computer systems management. In this paper, a year later, I will attempt to review for this workshop some of the more important new findings which other software scientists have reported, discuss their significance, and, if time permits, consider an area in which further work might be quite profitable.

Even though all of the work I will report on is rather intimately interrelated, it can conveniently be divided into a number of areas in which important progress has been described in sufficient detail that I can summarize it for you. Special attention will be given to the new results in the areas of programming rates, precision and accuracy of relations, error rates, and operator frequency distributions. For anyone unfamiliar with the basic software metrics and equations, either the monograph [HAL77] or the paper in last year's proceedings [HAL77b] provide adequate background, and the notation used there will be followed.

Programming Rates

In the December 1977 issue of the IBM Systems Journal Claude Walston and C. P. Felix presented a terse but highly significant extension "On Lines of Code and Programmer Productivity" [WAF77b] to their earlier paper on "A Method of Programming Measurement and Estimation" [WAF77]. In the extension Walston approximated the software science transcendental equations relating programming time to lines of source code by the expression

$$P = 1.27\ T^{2/3} \qquad (1)$$

where P is in thousands of source statements and T is in man-months. He then performed a least-squares fit to the 60 large systems in their IBM data base and obtained

$$P = 0.925\ T^{0.70} \qquad (2)$$

While the degree of agreement between the theory (Equation(1)) and experience (Equation(2)) is, in Walston's words, interestingly close, it takes a bit of pondering to realize the full significance of his finding. First, we must realize that his data base covers a range of programming projects of from less than 12 man-months to a maximum of 11,758 man-months. This suggests that the effect of job size on programming rates, not just total programming times, may be well defined by the data, as represented by equation (2).

Consequently, we can use either one of these equations to obtain the number of source lines per programmer per month for any given job size, where the job size can be expressed either in total lines of source code or in total implementation time. Using lines of source code, we have

Job Size	Source Lines per Man-Month	
(LOSC)	Theory	Observation
10,000	453	333
100,000	143	124
1,000,000	46	46

and using total man-months gives

Job Size	Source Lines per Man-Month	
(Man-Months)	Theory	Observation
12	555	439
120	257	220
1,200	120	110
12,000	55	55

Expressed in this way, it is apparent that we now nave, quantitatively, a theoretical, experimentally confirmed explanation for not just average programming times, but the decrease in hourly production rates accompanying increases in project size.

When one remembers that the effort relation was originally tested, and actually derived, with programming tasks requiring from 5 to 90 minutes, its extension to real world projects of this magnitude suggests that the most important variables have been identified, and their role, on the average or statistically speaking, is now understood.

From the engineering point of view there is little to choose between equation (1) and equation (2), because they both give about the same answers. But from the scientific point of view, the difference is fundamental. This is true because the theoretical equation was derived from first principles. Its coefficients depend only upon four measurable quantities: 1) the rate at which the brain can make elementary mental discriminations, or Stroud Number (18 e.m.d./sec.); 2) the average number of operators and operands in an executable statement (7.5); 3) the average fraction of total statements which are executable (1/2); and 4) the average language level (λ=1.34). Consequently, if any one of these parameters is changed, the effect on programming rates is calculable.

Precision and Accuracy

The question of how closely we should expect the various equations of software science to reflect individual programs has been, in principle, unanswered. Most of us working in the field have been subscribing to the view that software science is a bit like actuarial statistics. In that field, for example, one might find that men at age 65 have a life expectancy of 14 years. But this in no way guarantees that any particular 65 year old man will not be killed by a truck in the next hour. In other words, the accuracy of the actuarial prediction is completely adequate, but its precision is so poor that for any individual case it may be useless.

Now if we examine a recent study of 34 General Motors PL/I programs by Elshoff [ELS78], we see that the correlation between the observed lengths of programs and the lengths calculated from the vocabulary -length equation is a highly significant 0.988, and the error of the mean is only 6.5%, according to his Table 4. But if we examine the errors for individual programs in that table, we see that they average 13.1 per cent on an absolute basis. For most engineering or management purposes, frequently concerned with differences among programs ranging from 500 to 50,000 statements, given a 13 per cent error in an individual program is not too serious.

But from the scientific point of view such errors raise an interesting and perhaps important question. If we considered only one of Elshoff's 34 programs, would its deviation from the vocabulary-length relation be attributable merely to the way in which it had been written, or might it instead depend upon the task it was designed to perform? If it is the later, then the relation is indeed akin to actuarial statistics, devoid of causes, but if it is the former then it results only from "experimental error". This question could be resolved if each of the 34 different programs was independently rewritten ten or twenty times. Then, if the mean values for each of the 34 tasks showed the same average deviation, that deviation should be attributed to some unidentified characteristic of the task. On the other hand, if the process of averaging over many independent implementations of each task reduced the deviation, then we would be safe in concluding that the length-vocabulary relation is completely independent of the task. If, in addition, it could be shown that such an averaging procedure yields increasingly close agreement with other software relationships, then the "actuarial statistics" analogy would have to be abandoned.

While the cost of performing the experiment described above is prohibitive, a smaller but much more comprehensive study recently completed by Woodfield [WOO78] appears to settle the question in the affirmative.

Woodfield reports upon two small related programs, each written independently by 18 Fortran programmers. The first program had four input/output parameters, or $\eta_2^*=4$. The second program merely increased the problem dimensions by one, or $\eta_2^*=5$. The remarkable thing about Woodfield's study is that in both cases he was able to predict not only the observed length, but seven other characteristics of the programs from nothing but the two values, language level $\lambda=1$, $\eta_2^*=4,5$. The results taken from his Table 2 are as follows.

Property		Original Version	
		Predicted	Observed
η_1	Unique Operators	9.27	9.89±1.81
η_2	Unique Operands	8.84	8.72±1.18
η	Vocabulary	18.11	18.61±1.69
N	Length	57.57	61.83±7.72
V	Volume	241	261±39
L	Level	.0645	.0585±.0146
V*	Potential Volume	15.51	14.95±3.02
η_2^*	I/O Parameters	4	3.84±.76

Property		Extended Version	
		Predicted	Observed
η_1	Unique Operators	10.59	10.61±1.75
η_2	Unique Operands	13.11	12.44±2.83
η	Vocabulary	23.70	23.06±2.98
N	Length	84.53	95.89±20.95
V	Volume	386	436±108
L	Level	.0509	.0481±.0117
V*	Potential Volume	19.65	20.77±6.85
η_2^*	I/O Parameters	5	5.21±1.58

In considering the agreement between the values predicted by theory and those observed, Woodfield notes that in no case was the discrepancy as great as six-tenths of the standard deviation, and that the average error of the 16 predictions was only 1.4 per cent.

In light of these results it seems safe to assume, for small programs at least, that deviations between observation and theory can be attributed to sampling error, rather than to unidentified characteristics of the programs themselves.

Error Rates

Linda Ottenstein has recently completed a comprehensive study [OTT78] relating programming error rates to the software parameters. While she analyzes a number of sets of data, including studies of her own, perhaps the most significant finding concerns the data of Lipow and Thayer [LIT77]. She derives the predicted number of "Validation" bugs, $\hat{B}_V$ as

$$\hat{B}_V = V^*/LE_0 = V/3000 \qquad (3)$$

where V is the volume, V* is the potential volume, L is the implementation level and the constant $E_0=3000$ is obtained directly from the psychological concept of chunking. Applied to the functionally grouped data of [LIT77], this gives, according to her Table 2.3.

Routine	Delivered Bugs	
	Observed	Predicted
	B_V	$\hat{B}_V$
A1	26	45
A2	67	63
A3	54	62
A4	41	47
A5	79	121
B1	105	66
B2	95	78
C1	239	221
C2	69	105
C3	55	33
C4	27	20
C5	50	70
C6	48	52
D1	87	180
D2	13	20
E1	144	136
F1	4	10
F2	8	24
F3	8	28
F4	30	32
F5	30	58
G1	238	241
G2	22	29
H1	1	7
H2	466	406

Coef. of Corr. r = 0.962

The agreement between observation and theory is such that all but 7½ per cent ($1-r^2$) of the variance is explained by it. But because the error or "problem report" data in [LIT77] were available both in the 25 functional groupings above, and also in terms of some 250 individual procedures [THA76] making up these 25 functional groups, Ottenstein was able to test facet of equation (3). According to its derivation, while the number of errors must increase with the size of a program, the rate at which they are made must decrease as the programmer gains familiarity with the program on which he is working. Consequently, if we considered two subprocedures of equal volumes, we should expect different error rates depending upon the order in which they were written. For the total function, only the total volume would be involved, but for a sub-function, its implementation order should contribute to its variance. The theory therefore predicts that the correlation between B_V and $\hat{B}_V$ for the 250 sub-procedures should not be near one, but near the square root of one-half or 0.71. Performing the analysis, she found a value of r=0.756, rather than the r=0.962 for the functionally grouped units.

This gives further credence to her derived expressions for the average time required to find and correct one "validation bug",

$$T_V{}^1 = \frac{1}{4} \times \frac{3000}{LS} \qquad (4)$$

and for the average number of required runs per day per programmer during validation

$$R_V/\text{day} = 48SL \qquad (5)$$

where L is the implementation level and S is the Stroud Number.

Operator Frequency Distributions

A number of research workers have been intrigued by the uniform pattern observed whenever the frequency of occurrence of individual operators in a program are rank-ordered, and a series of papers have resulted from their attempts to derive a mathematical expression for the observed frequency distribution. One of several motivations for this work lies in the possibility that the results would be useful in resolving ambiguities in the classification of operators in new languages.

In a recently submitted paper, Zweben [ZHE78] has demonstrated that the relationship obeys the equation

$$i = X_0 \log_2 (X_0/X_i) \qquad (6)$$

where i is the rank-order, starting from zero for the most frequently occurring operator, and

$$X_i = \log_2 f_i \qquad (7)$$

where f_i is the number of occurrences of the i th most frequent operator. Since the total number of operators N_1 must be given by

$$N_1 = \sum_{i=0}^{\eta_1 - 1} f_i \qquad (8)$$

it is possible to solve for the expected number of occurrences f_i for any value of i, given only η_1 and N_1. For 34 PL/I programs, they report a mean correlation coefficient of 0.989, and a line of regression with a slope of 0.984 and an intercept of -0.28.

Missing Data

There are still more areas in software science needing study than there are presently being investigated. Basic issues, such as the role of modularity, application to top-down design, and to learning theory may yet require years of effort. But in considering the specific field of Life Cycle Management, it is easy to note that one simple tool is missing. No one has yet implemented a software parameter analyzer for Cobol. Consequently, we have mean values, and variances, of the language levels for Fortran, PL/I, Algol and others, but not for Cobol.

References

[ELS78] Elshoff, James L., "An Investigation into the Effects of Counting Methods Used on Software Science Measurements", ACM SIGPLAN Notices, Vol. 13, No. 2, February 1978, pp. 30-45.

[HAL77] Halstead, M. H., Elements of Software Science, Elsevier North Holland, New York, 1977.

[HAL77b] Halstead, M. H., "Potential Impacts of Software Science on Software Life Cycle Management", in Software Phenomenology, U.S. Army Institute for Research in Management Information and Computer Science, August 1977, pp. 385-400.

[LIT77] Lipow, M., and T. A. Thayer, "Prediction of Software Failures" Proceedings of the 1977 Annual Reliability and Maintainability Symposium, January 18-20, 1977.

[OTT78] Ottenstein, Linda M., "Predicting Parameters of the Software Validation Effort", Ph.D. Thesis, Purdue University, August 1978.

[THA75] Thayer, T. A., "Understanding Software through Empirical Reliability Analysis", AFIPS Conference Proceedings, Vol. 44, 1975, pp 335-341.

[WAF77] Walston, C. E. and C. P. Felix, "A Method of Programming Measurement and Estimation", IBM Systems Journal, Vol. 16, No. 1, pp. 54-73.

[WAF77b] Walston, C. E. and C.P. Felix,

"on Lines of Code and Programm-
Productivity," IBM Systems
Journal, Vol. 16, No. 4, 1977,
pp. 421-423.

[WOO78] Woodfield, Scott, "An Experiment on Unit Increase in Algorithm Complexity", Submitted

[ZHE78] Zweben, S. H., M. H. Halstead and James L. Elshoff, "The Frequency Distribution of Operators in PL/I Programs" Submitted.

COST EFFECTIVENESS IN SOFTWARE ERROR ANALYSIS SYSTEMS

Mary Anne Herndon

Herndon Science & Software
San Diego, CA

ABSTRACT

Software error analysis systems must have the capability of functioning as both a cost effective and valuable managerial tool. To achieve this capability, the design of the data collection must reflect the individual project's managerial concerns, and the resulting empirical analysis should be available for long term access.

INTRODUCTION

The implementation of software error analysis systems has resulted in enormous expenditures for data collection and analysis. Although many researchers have reported empirical analyses of these data (1,2,3), few important managerial problems have been addressed and solved. Researchers have analyzed the data to provide descriptors, such as estimates of the mean time between failures, and, to a limited extent, error classifications. Effective project management, however, demands more information than, for example, an estimate of mean time between failures, or a tally of the number of register misassignments. As yet, neither the expense of collecting software error data nor the personnel difficulties encountered have been justified by the usefulness of the information obtained.

There is a variety of ways to collect immense amounts of different types of software error data. In response to key project managerial demands, however, relevant data are currently being ignored. In order to make software error analysis systems both a valuable and cost effective managerial tool, attention needs to be focused on the relevant issues in project management.

DATA COLLECTION DESIGN

The selection of data to be collected during validation should be tailored to the management goals of each project. Key management personnel, as well as in-house reliability personnel, should be consulted in the selection of data items. Although each project requires different types of information, still, there are managerial concerns that are crucial to all types of software projects. The first concern centers around the utilization of expenditures allocated for validation. It is a well known fact that a large portion of total project expenditures go into formal validation, even as much as 50% to 75%, depending upon the type of project. However, the accounting of total expenditures is usually not performed until test completion. A daily or session accounting of expenditures provides for more effective monitoring of the progress of validation than a final accounting. In designing the types of information for monitoring the apportionment of expenditures, the following items are quite useful in current and predictive measures of expenditure utilization.

1. A cost categorization for errors, such as proposed by Herndon and Keenan (4).,

2. A tallying of cost types of errors detected during each test session.

From this minimal amount of data, managers are able to obtain intuitive information on the cost effectiveness and error detection effectiveness of the validation procedure.

The second managerial concern centers around providing information describing the dynamics of the functioning system during validation. Complicated software systems, such as exemplified by real time systems, need to be understood in terms of module interactions. Although modern program design philosophy attempts to minimize the extent such interactions, in the design of actual systems this is not always achieved. Consequently, errors that appear are not always localized to the module of occurrence. This type of information provides useful diagnostic information in the event of serious operational problems. In one empirical study of a real time system (5), a seemingly innocous module contained an inordinate number of high impact errors. The following minimal set of data is suggested to obtain system interaction profiles.

1. Error impact classification categories, such as those proposed by Herndon, et al.

2. Profiles of each system component for each category.

3. Calculation of an average impact for each module.

Observations from this empirical analysis can indicate potentially troublesome or unstable area of the system. For example, if one particular module seems to be associated with a large number of high impact failures, redesign of the module might be advantageous before the product is released. With the proper identification of unstable modules, appropriate fault tolerant measures can be used for compensation.

RECOMMENDATIONS

In surveying the current state of the art of software error analysis systems, two glaring deficiencies are evident. The first deficiency concerns the choice of data that is selected for collection. Software error analysis systems must function as a cost effective managerial tool in order to compensate for the overhead that is associated with data collection. As of now, there is a distinct impression of quantity of data items as opposed to relevancy. Cost effectiveness in software error analysis systems can only be achieved by the proper design of a minimal set of information to answer specific managerial questions.

The second deficiency concerns the short-lived usefulness and availability of the data after the product is released. The empirical analysis of the reliability data that is obtained during validation provides an initial system reliability profile. As the product ages and requires alterations, the information gained from validation is crucial for effective field management.

REFERENCES

(1) J.D. Musa, A theory of software reliability and its application," IEEE Trans. on Software Engineering, Vol. SE-1, No. 3, September 1975, pp. 312-327.

(2) M. Shooman, "Structural models for software reliability prediction," in Proc. 2nd International Conference on Software Engineering, San Francisco, Oct. 1976, pp. 268-280.

(3) H. Hecht, "Reliability Measurement During Software Engineering, San Francisco, Oct. 1976, pp. 268-280.

(4) M.A. Herndon and J.A. Lane, "Analysis of Software Errors for Cost Factors," Proceedings of 1977 AIAA Computers of Aerospace Conference, Los Anges, CA, pp. 455-459

(5) M.A. Herndon and A.P. Keenan, "Analysis of Error Remediation Expenditures During Validation," Proceedings of the 3rd International Conference on Software Engineering, Atlanta, GA, 1978, pp. 202-206.

STATISTICAL TECHNIQUES FOR COMPARISON OF COMPUTER PERFORMANCE

Sandra A. Mamrak
The Ohio State University
Columbus, Ohio 43210

Abstract

The comparison of computer performance requires a methodology designed to lead to the selection of the best computer and to provide control of the probability of having made a correct choice. Methodologies often used in classical statisical designs lead to regression analyses of the data, employing either analysis of variance or curve-fitting techniques. The questions that can be answered using such methodologies are of the type "Is the performance of several alternative systems the same (are the distributions of performance measurements identical from a statisical point of view)?" or "How does one particular system performance parameter depend upon the other system parameters?". In most computer comparison efforts, however, these questions are not appropriate. The question of real interest is: "Which system is the best?" or "How do the systems rank from best to worst?" It is precisely for this type of problem that statistical ranking and selection procedures were developed. These procedures are applied in the selection methodology outlined below.

A Computer Selection Model

A model of the way computer selection is typically done focuses the main components of the computer selection process and provides a framework for this paper. The model is presented in Figure 1 and discussed in the next two sections.

Classification of Performance Criteria

In choosing the best alternative from several alternatives, criteria must be defined which state what is meant by best. These criteria can be categorized in two ways. They are either measurable or nonmeasurable, and they are either mandatory or desirable. Thus, the selection criteria can be classified as Mandatory Nonmeasurable (MN), Mandatory Measurable (MM), Desirable Nonmeasurable (DN), and Desirable Measurable (DM). Examples of each type of criterion are provided in Table 1.

Application of Performance Criteria

Figure 1 illustrates a sequence in which the classes of selection criteria are applied in the process of choosing the best alternative system. This sequence is composed of three phases. The application of MN criteria in phase I is easily managed, since each alternative either does or does not have the required characteristic. Phase II involves the application of MM criteria. Experiments are conducted on the alternatives which survived Phase I and the performance of each is documented. In general, for each MM criterion, measurements are gathered from every system and a decision is made as to whether or not the criteria are satisfied. Failure to satisfy a single MM criterion results in an alterntiave's elimination. At the conclusion of Phase II, the number of alternatives is usually reduced from that of Phase I.

Finally, determination of the best alternative is made in Phase III. This stage is separated into two parts, Phase IIIA for the application of DN criteria and stage IIIB for the application of DM criteria. For DM criteria, data are collected from each alternative being evaluated and compared. On the basis of relative performance with respect to the DN and DM criteria, an alternative is selected as the best. In both Phases II and IIIB, comparison requires collecting and analyzing relevant performance measurements for the various computer alternatives under consideration: in Phase II to select those satisfying certain mandatory performance standards and in Phase IIIB, to select the best remaining one.

A selection process requires a methodology to lead to the selection of better than standard alternatives and selection of the best alternative and to simultaneously provide control of the probability of having made correct choices. This work focuses its attention on the specification of a methodology for those problems which occur in Phase II and IIIB, thereby leading to a more comprehensive, scientific methodology for computer selection.

A Computer Selection Methodology

A good experimental design is a critical component of any comparison methodology, since the efficiency of the data collection process and the validity of the data analysis depend upon it. Ranking and selection procedures (see [KLE75] or [GIB77] for a survey of these techniques) provide an appropriate experimental design for computer selection. These procedures can be roughly characterized as following three lines of development: one set of procedures ranks systems by comparing sample means,

Figure 1. Model of the Computer Selection Process

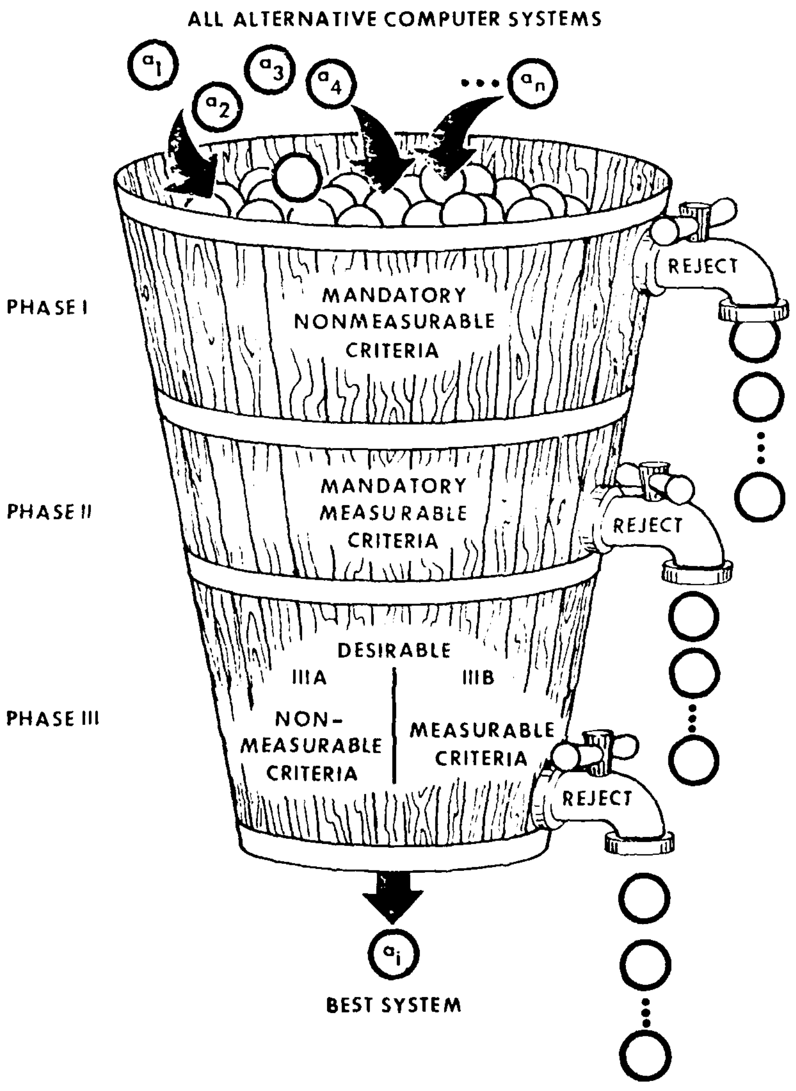


Table 1. EXAMPLE OF PERFORMANCE CRITERIA

Type	Example
Mandatory Nonmeasurable	1. The system must be fully delivered and operational no later than September 1, 1979.
	2. Timesharing service must include FORTRAN, Basic, Lisp, SNOBOL and editing facilities.
Mandatory Measurable	1. The mean-time-to-failure for a specific one month period must be greater than 4 hours.
	2. 95% of all trivial command response times must be less than 1 second.
Desirable Nonmeasurable	1. It is desirable that the system include Pascal and COBOL facilities.
	2. It is desired that the system cost less than $100,000.
Desirable Measurable	1. It is desired that the system provide as fast a mean turnaround time as possible for the benchmark run.
	2. It is desired that response time means as well as variations be small.

one by comparing sample percentiles and one by comparing sample proportions. In all three cases, the procedures specify the number of data points which must be collected from each system in a comparison study in order to guarantee that the probability of a correct selection be greater than or equal to a predetermined minimum value.

The use of a mean, percentile or proportion statistic for system comparison is an analyst choice based on considerations about the objectives of a comparison experiment, the statistical properties of the data and the statistical requirements of the data analysis techniques. Means are often used for comparisons when criteria like script turnaround time or script cost are of interest. For comparison criteria such as response time, which tend to have exponential-like distributions, the mean is not as meaningful a statistic. Percentages or proportions are more appropriate.

The percentile and proportion approaches to comparison are very similar in that they both rely on a single comparison criterion's cumulative distribution. The difference lies in whether an analyst prespecifies a desired percentage value or a desired comparison criterion value. In a comparison based on percentiles, a percentage is predetermined. Results are produced of the form:

"if computer service A has 90% of its response times less than 3 seconds, and computer service B has 90% of its response times less than 3.5 seconds, then rank A as being better than B,"

where "90%" is prespecified by the analyst. In a comparison based on proportions, a value of a comparison criterion is prespecified. Results are produced of the form:

"if computer service A has 80% of its response times less than 3 seconds, and computer service B has 87% of its response times less than 3 seconds, then rank B as being better than A,"

where "3 seconds" is prespecified by the analyst.

Selection of Systems Better than a Standard

In the case of selecting those computer systems which are better than a standard (Phase II in Figure 1), an experimental design is required which leads to an analysis of the data that answers the question "Which services are at least as good as a prespecified standard?". The ranking and selection techniques which have been developed for selection better than a standard are not appropriate for computer selection when performance indices are being compared at their mean or percentile values. The procedures make assumptions about the data that are clearly not justified in computer selection experiments (such as equal variance for all systems). But, an appropriate procedure does exist when proportions are the basis for a selection (see [MAM78]).

Selection of the Best System

In the case of selecting the best computer service (Phase III in Figure 1), an experimental design is required which leads to an analysis of the data that answers the question "Which system is the best one?". In this case ranking and selection techniques exist for choosing systems based on mean or percentile values (see [AME78b] and [MAM77]), and also based on proportions (see [AME78a] and [MAM78]).

A Feasibility Study

A large scale feasibility case study is underway to evaluate the time and cost required to apply the computer service comparison methodology in an actual procurement environment. Four heterogeneous, remote-access, time sharing services are being compared. The specifications for the case study and the experimental results are presented in [MAM78].

References

AME78a Amer, P. D., "Experimental Design in Computer Comparison and Selection," Ph.D. Dissertation, Dept. of Computer and Information Science, The Ohio State University, December 1978.

AME78b Amer, P. D. and S. A. Mamrak, "Statistical Methods in Computer Performance Evaluation: A Binomial Approach to the Comparison Problem," Proceedings Computer Science and Statistics: Eleventh Annual Symposium on the Interface, Institute of Statistics, North Carolina State University, March 1978, pp. 314-319.

MAM77 Mamrak, S. A. and P. A. DeRuyter, "Statistical Methods for Comparison of Computer Services," Computer, Vol. 10, No. 11, November 1977, pp. 32-39.

MAM78 Mamrak, S. A. and P. D. Amer, "A Methodology for the Selection of a Computer Service," NBS Special Publication, in preparation, 1978.

SOFTWARE COMPLEXITY MEASUREMENT

Thomas J. McCabe

Independent Consultant
5380 Mad River Lane
Columbia, Maryland 21044

This paper describes a graph-theoretic software complexity measure, and describes how it can be applied to limit the logic in a module during the design stage so it is testable and maintainable during later stages.

The process of software construction relies in a fundamental sense on the sizing of subsystems and modules that are separately designed, coded and tested. One concept very sorely needed is a way to measure and control software complexity at the design stage in such a fashion that the modules are subsequently manageable, i.e., comprehensible, and also testable. The approach of this research is to develop a concrete, applicable, but yet mathematically based measurement of software complexity. The measure presented in this paper has a mathematical basis in the sense that it is not dependent on the application a module is used for, the language used, or the programming style--it measures purely the complexity of internal logic of a module. The measure presented is also immediately applicable because it is readily understood by the programming community and it directly mapps into a rigorous testing methodology and makes maintenance feasible.

The complexity measure decided upon will limit the number of independent paths in a program at the design stage so the testing will be manageable during later stages. One of the reasons for limiting independent paths instead of limiting all potential paths is the following dilemma: "A relatively simple program can have an arbitrary high number of paths." The approach taken here is to limit the number of basic (or independent) paths that when taken in combination will generate all paths.

Definition

One definition and one theorem from graph theory will be needed to develop these concepts. See Berge [1] for a reference.

Definition 1

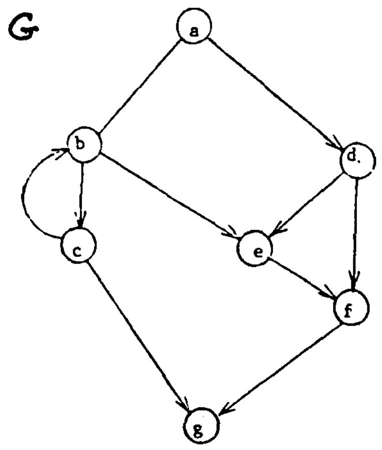
The cyclomatic number $v(G)$ of a graph G within n vertices, e edges and p connected components is $v(G)=e-n+p$.

Theorem 1

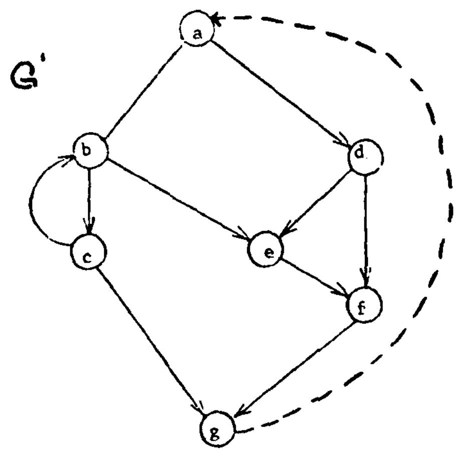
In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

The application to computer programs will be made as follows. Given a program associate with it a graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of statements where the flow is sequential and the arcs represent the program's branches taken between blocks. This graph is classically known as the control graph (see Ledyard [12]) and it is assumed that each node can be reached by the entry node and each node can reach the exit.

For example the control graph shown below has seven blocks (a) through (g), entry and exit nodes (a) and (g), and ten arcs.



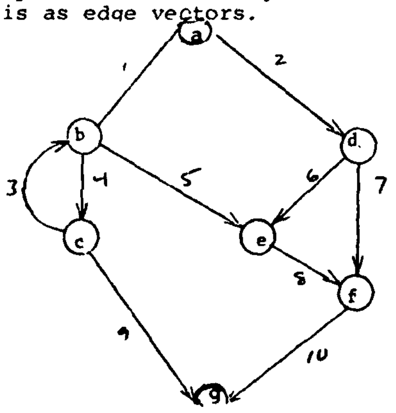
In order to apply Theorem 1 the graph must be strongly connected which means that given two nodes (a) and (b) there exists a path from (a) to (b) and a path from (b) to (a). To satisfy this we associate an additional edge with the graph which branches from the exit node (g) to the entry node (a) as shown below.



Theorem 1 now applies and it states that the maximal number of independent circuits in G^1 is 11-7+1 (G has only one connected component so we set p equal to 1. The generalized case where $p>1$ is used for design complexity, (see McCabe [11]). The implication therefore is that there is a basis set of 5 independent paths that when taken in combination will generate all paths. For example, the set of 5 paths shown below form a basis.

b1:	abcg
b2:	$a(bc)^2g$
b3:	abefg
b4:	adefg
b5:	adfg

If one chooses any arbitrary path it should be equal to a linear combination of the basis paths b1-b5. For example, the path abcbefg is equal to b2+b3-b1, and path $a(bc)^3g$ equals 2*b2-b1. To see this it is necessary to number the edges in G and show the basis as edge vectors.



Basis

	1	2	3	4	5	6	7	8	9	10
b1	1	0	0	1	0	0	0	0	1	0
b2	1	0	1	2	0	0	0	0	1	0
b3	1	0	0	0	1	0	0	1	0	1
b4	0	1	0	0	0	1	0	1	0	1
b5	0	1	0	0	0	0	1	0	0	1
abcdefg	1	0	1	1	1	0	0	1	0	1
$a(bc)^3g$	1	0	2	3	0	0	0	0	1	0

The path abcbefg is represented as the edge vector shown above and it is equal to b2+b3-b1 where the addition is done component-wise. In similar fashion the path $a(bc)^3g$ shown above is equal to 2*b2-b1.

It is important to notice that Theorem 1 states that G has a basis set of size 5 but it does not tell us which particular set of 5 paths to choose. For example, the following set will also form a basis.

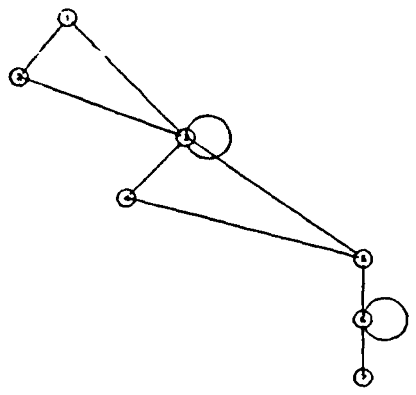
adfh
abefg
adefg
$a(bc)^3befg$
$a(bc)^4g$

When this is applied to testing the actual set of 5 paths used will be dictated by the data conditions at the various decisions in the program but the theorem guarantees that we will always be able to find a set of 5.

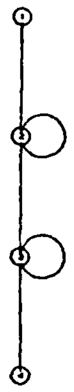
It should be emphasized that the process of adding the extra edge to G was only to make the graph strongly connected so Theorem 1 would apply. When calculating the complexity of a program or testing the program the extra edge is not an issue but rather it is reflected in the expression used for complexity. The complexity v, therefore, is defined as v=e-n+2p since an extra edge is added for each component. In each of the examples discussed in this paper there is only one component so the complexity expression simplifies to v=e-n+2.

Examples

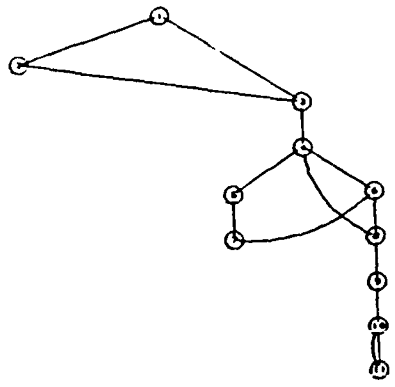
Several actual control graphs and their complexity will be presented in order to illustrate these concepts. These graphs came from FORTRAN programs on a PDP-10. The programs were analyzed by an automated system FLOW that recognizes the blocks and transitions in a FORTRAN program, computes the complexity, and draws the control graphs on a DATA DISC CRT. The straight edges represent downward flow (e.g. in the second graph below the line between (2) and (3) means that (2) branches to (3). The curved arcs represent backward branches (e.g. in the second graph (5) branches back to (2)).



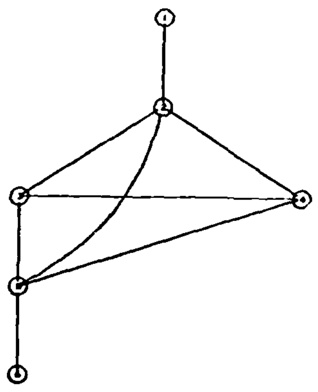
v=5



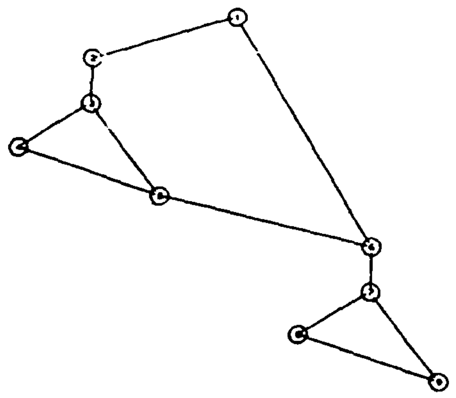
v=3



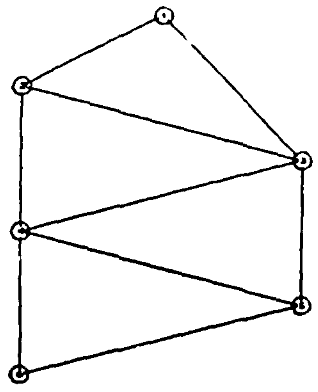
v=5



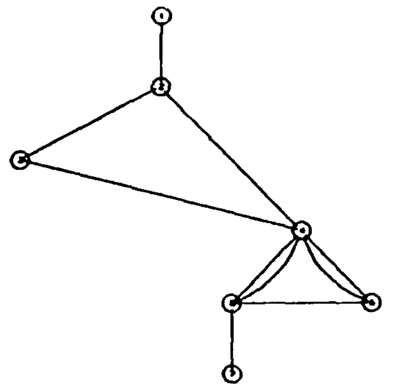
v=4

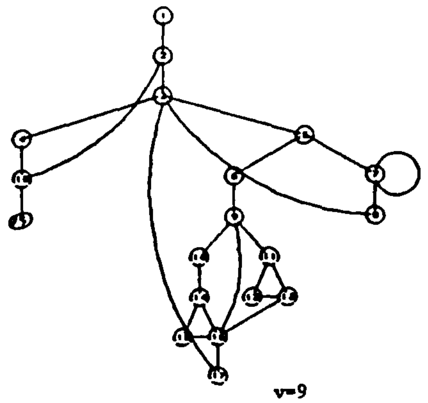


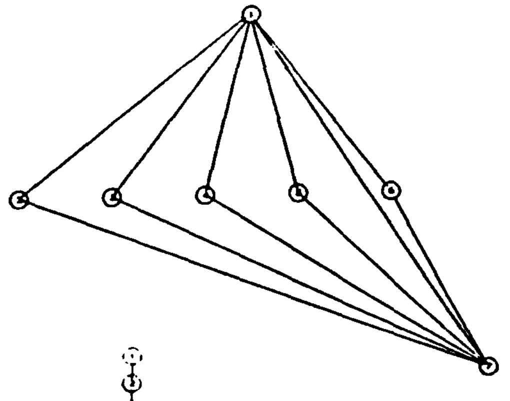
v=4

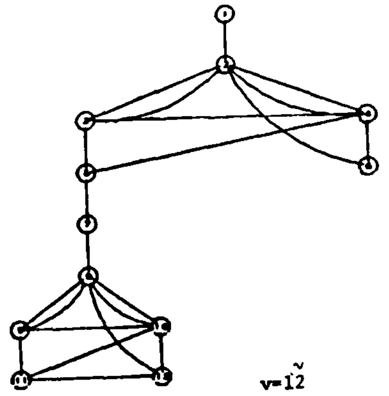


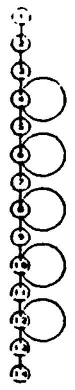
v=5

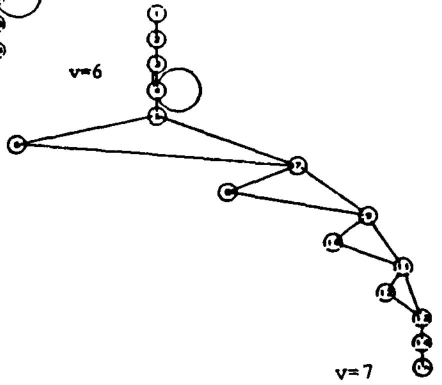

v=5


v=9

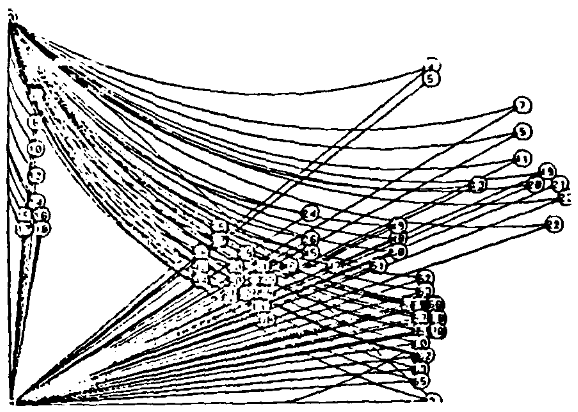

v=6


v=12


v=6


v=7

The graphs above were presented in order of increasing complexity in order to suggest the relationship between the complexity numbers and our intuitive notion of the complexity of the graphs. One essential ingredient in any testing methodology is to limit the program logic during development in order that, first, the program can be understood, and second, the amount of testing required to verify the logic is not overwhelming. These techniques are being presented in The Institute for Advanced Technology's Structured Testing seminars and experience has consistently shown that programs with high cyclomatic complexity are difficult to understand and are rarely tested adequately. For example, according to its author the program below is 'one of my better programs' and it required only 'about four or five tests to verify'.



v=47

The physical size of the program this graph is derived from is only 70 lines of source code. The physical size of several of the twelve previous graphs exceeded 70 lines, and, in fact, only a weak correlation between physical size and complexity has been found. Because of this, the common practice of attempting to limit complexity by only controlling how many pages a routine will occupy is entirely inadequate. This complexity measure has been used in production environments by limiting the complexity of every module to 10. Programmers have been required to calculate the complexity as they develop routines, and if it exceeds 10 they are required to recognize and modularize subfunctions or re-design the software. The only situation where the limit of 10 seemed unreasonable and an exception was allowed is in a large CASE statement where a number of independent blocks followed a selection function.

See McCabe [11] for further simplification of this research. In particular, reference [11] contains results that allow a simplification of the computation of e-n+2p into more intuitive programming terms. Reference [11] also comments on the induced testing methodology and it deals with the measurement of structuredness of programming logic.

References

1. C. Berg, Graphs and Hypergraphs. Amsterdam, The Netherlands: North Holland 1973.

2. B.W. Boehm, "Software and its impact: A Quantitative Assessment", Datamation, vol. 19, pp. 48-59, May 1973.

3. W.B. Cammack and H.J. Rogers, "Improving the Programming Process", IBM Tech. Rep. TR 00.2483, Oct. 1973.

4. J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Test Data Selection", Proc, 1975 International Conf. on Reliable Software, Los Angeles, California, pp. 493-510.

5. W.C. Hetzel, "Program Test Methods: Prentice Hall, Inc.", Englewood Cliffs, New Jersey, 1973.

6. W.E. Howden, "Symbolic Testing and the Dissect Symbolic Evaluation System", IEEE Trans. on Software Engineering Vol. SE-3 No. 4, pp. 266-278, July 1977.

7. J.C.King, "A New Approach to Program Testing", Proc., 1975 International Conf. on Reliable Software, Los Angeles, California, pp. 228-233.

8. D.E.Knuth, "Structured Programming with GOTO Statements", Computing Surveys, vol. 6, pp. 261-301, Dec. 1974.

9. R. Kosaraju, "Analysis of Structured Programs", J. Comput. Syst. Scil, vol. 9, pp. 232-255, Dec 1974; also Dep. Elec Eng., The Johns Hopkins Univ., Baltimore, Md., Tech. Rep. 72:11, 1972.

10. H. Legard and M. Marcotty, "A Generalogy of Control Structures", Commun. Assoc. Comput. Mach., Vol. 18, pp. 629-639, Nov. 1975.

11. T.J. McCabe, "A Complexity Measure", IEEE Trans. on Software Engineering, Vol. SE-2 No. 4, pp. 308-320, Dec 1976.

12. E.F. Miller, "Program Testing: Art Meets Theory", Computer, Vol. 10, No. 7, pp. 42-51, July 1977.

13. H.D. Mills, "Mathematical Foundations for Structured Programming", Federal System Division, IBM Corp, Gaithersburg, Md., FSC 72-6012, 1972.

14. J.F. Sullivan, "Measuring the Complexity of Computer Software", MITRE Corporation Report, 1973.

THE UTILITY OF SOFTWARE QUALITY METRICS IN LARGE-SCALE SOFTWARE SYSTEM DEVELOPMENTS

JAMES A. McCALL

GENERAL ELECTRIC COMPANY
SUNNYVALE, CALIFORNIA

ABSTRACT

This paper describes the utility of the progressive application of software quality metrics during large-scale software developments. The concept of the software quality metrics was derived and validated during a study supported by the Air Force Systems Command Electronic Systems Command and Rome Air Development Center. Further extensions to the concepts are currently being supported by Rome Air Development Center and the U.S. Army Computer Systems Command, AIRMICS. The metrics provide a disciplined, engineering approach and life cycle management viewpoint to software quality assurance.

INTRODUCTION

In a study for the Air Force, a program management-oriented view of software quality was established (Ref 1). The concept of quality derived was based on three viewpoints with which a program manager interacts with the end product: Its operation, revision, and transition. These viewpoints correspond to the life cycle activities of the product. The factors of software quality associated with these three viewpoints are shown in figure 1.

These factors are used by the program manager to identify which qualities are important to the particular development effort. The desired level of quality can be quantitatively specified based on the formal definitions established for these factors.

Associated with each quality factor is a set of criteria. These criteria are attributes of the software and related documentation and their presence provides the quality or characteristics implied by the factor. Software metrics have been established which provide quantitative measures of those attributes represented by the criteria. The metrics allow measurement of the progress toward achieving desired levels of quality by their application to intermediate products produced during a large-scale software development.

These concepts are extensions of some significant efforts by others in this field (Refs 2, 3, 4, 5, 6). We have added a program manager's orientation, automated the collection of many of the metrics, and added quantification to measures applied during the early phases of a development.

The concept is based on the following facts about quality:

- Relative to the application
- Impacts the cost to develop

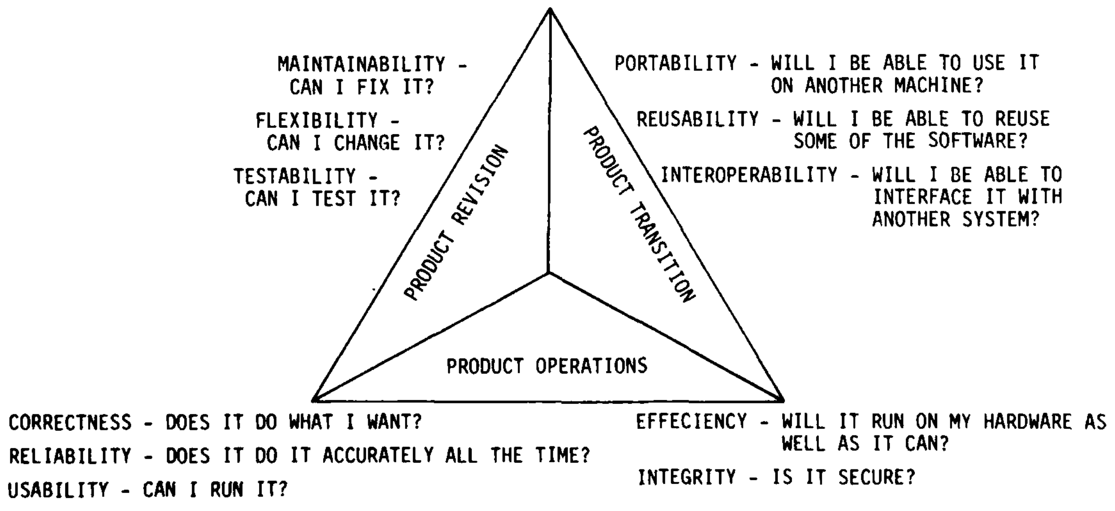

MAINTAINABILITY - CAN I FIX IT?
FLEXIBILITY - CAN I CHANGE IT?
TESTABILITY - CAN I TEST IT?
PRODUCT REVISION
PRODUCT TRANSITION
PORTABILITY - WILL I BE ABLE TO USE IT ON ANOTHER MACHINE?
REUSABILITY - WILL I BE ABLE TO REUSE SOME OF THE SOFTWARE?
INTEROPERABILITY - WILL I BE ABLE TO INTERFACE IT WITH ANOTHER SYSTEM?
PRODUCT OPERATIONS
CORRECTNESS - DOES IT DO WHAT I WANT?
RELIABILITY - DOES IT DO IT ACCURATELY ALL THE TIME?
USABILITY - CAN I RUN IT?
EFFECIENCY - WILL IT RUN ON MY HARDWARE AS WELL AS IT CAN?
INTEGRITY - IS IT SECURE?

Figure 1. Factors in Software Quality

- Impacts the life cycle costs
- Should be customer-defined
- Has an associated cost to measure and control.
- Its measurement will change as the technologies of producing software change.

The fact that there is a trend toward more structured disciplined programming techniques and environments reinforces the concept of software quality metrics.

The purpose of this paper is to expand upon the utility of the periodic application (measurement) of software quality metrics during large-scale software developments.

A MEASUREMENT VEHICLE

At appropriate times during a large-scale development, the application of the metrics results in a matrix of measurements. The metrics that have been established to date are at two levels - system level and module level. The approach to be described is applicable to both levels and will be described in relationship to the module level metrics.

A n by k matrix of measurements results from the application of the metrics to the existing products of the development (e.g., at design, the products might include review material, design specifications, test plans, etc.) where there are k modules and n module level measurements applicable at this particular time.

$$M_d^m = \begin{bmatrix} m_{11} & m_{12} & \cdot & \cdot & \cdot & m_{1k} \\ m_{21} & & \cdot & & & \\ \vdots & & & \cdot & & \\ m_{n1} & & & & \cdot & m_{nk} \end{bmatrix}$$

For that particular time there is an associated matrix of coefficients which represent the results of linear multivariate regression analyses against empirical data (past software developments). These coefficients, when multiplied by the measurement matrix results in an evaluation (prediction) of the quality of the product based on the development to date. This coefficient matrix, shown below, has n columns for the coefficients of the various metrics and 11 rows for the 11 quality factors.

$$C_d^m = \begin{bmatrix} c_{11} & c_{12} & \cdot & \cdot & \cdot & c_{1n} \\ \cdot & \cdot & & & & \\ \cdot & & \cdot & & & \\ \cdot & & & \cdot & & \\ c_{11,1} & & & & & c_{11,n} \end{bmatrix}$$

To evaluate the current degree or level of a particular quality factor, i, for a module, j, the particular column in the measurement matrix is multiplied by the row in the coefficient matrix. The resultant value:

$$c_{i,1}\, m_{i,j} + c_{i,2}\, m_{2,j} \cdot \cdot \cdot + c_{i,n}\, m_{n,j} = r_{i,j}$$

is the current predicted rating of that module, j, for the quality factor, i.

The coefficient matrix should be relatively sparce (many $c_{i,j} = 0$). Only subsets of the entire set of metrics applicable at any one time relate to the criteria of any particular quality factor.

Multiplying the complete measurement matrix by the coefficient matrix results in a ratings matrix. This matrix contains the current predicted ratings of each module for each quality factor.

$$CM = R_d^m = \begin{bmatrix} r_{11} & r_{12} & \cdot & \cdot & \cdot & r_{1,k} \\ \cdot & \cdot & & & & \\ \cdot & & \cdot & & & \\ \cdot & & & & \cdot & \cdot \\ r_{11,1} & & & & & r_{11,k} \end{bmatrix}$$

This approach represents the most formal approach to evaluating the quality of a product utilizing the software quality metrics. Because the coefficient matrix has been developed only for a limited sample in a particular environment, it is neither generally applicable nor has statistical confidence in its values been achieved.

Other valuable information is available from the measurement matrix with the current state of the technique.

For example, the development of a particular module can be assessed by examining the measures in the appropriate column of the measurement matrix. How all the modules are doing with respect to any particular attribute can be assessed by examining the appropriate row. Those particular modules with excessively low scores should be investigated further. This form of sensitivity analysis is facilitated by the collection of the metric data in the form of the measurement matrix.

In examining a particular measure across all modules, consistently low scores may exist. This situation identifies the need for a new standard or stricter enforcement of existing standards to improve the overall development effort.

As experience is gained with the metrics and data is accumulated, threshold values or industry acceptable limits may be established. In addition, comparisons of metric values with trends in the occurrence of Design Problem Reports and Software Problem Reports may prove to provide important quality assurance insight.

QUALITY CONTROL MECHANISM

The periodic application of the metrics during a large-scale software development can be viewed as a control system. Snapshot assessments are generated, feedback to program management is provided with respect to their specified requirements for quality, thereby allowing corrective action, calibration, redirection, or the identification of areas to be emphasized later in the development (e.g., testing) to be enacted.

These concepts are illustrated in figure 2. It is important to note, the metrics were established with a goal of not requiring additional products to be generated during the development efforts. Thus, the metrics take advantage of current control mechanisms (delivered products and reviews) normally utilized in a large software development.

EXPERIENCES

These concepts have been applied to software developed in a command and control system environment. This application resulted in the current coefficient matrices and sets of metrics to be applied during a software development (Ref 7, 8).

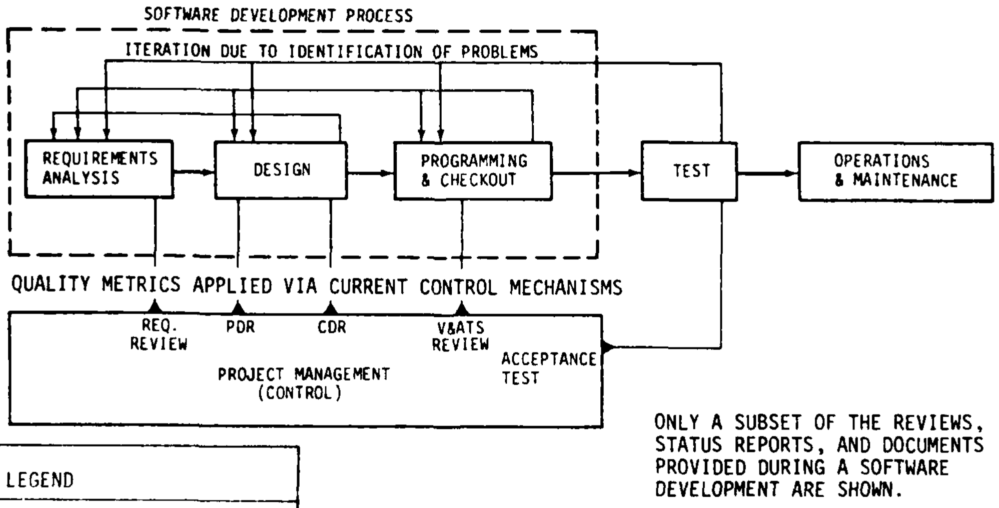
We are currently assessing the value of these concepts to the development and modification of support software. Also, we have just begun an effort with Air Force and Army sponsorship (Ref 9) to evaluate the metrics with respect to an Army management information systems environment. The experience gained from these efforts will provide an excellent basis for discerning the extent to which these concepts can provide additional quality assurance discipline in large-scale software developments.

In effect, these concepts are an attempt to provide a mechanism for the specification of life cycle-related quality goals and assessment of the progress toward those goals during the early development phases. The quantification provided by the metrics affords more consistent evaluation of the software quality. The overall goal in these efforts is to introduce a more disciplined, engineering approach and a life cycle management viewpoint to software quality assurance.

REFERENCES

1. J. McCall, P. Richards, G. Walters, "Factors in Software Quality, " three volumes, NTIS AD-A049-014, AD-A049-015, AD-A049-055, Nov. 1977. (Provided under contract F30602-76-C-0417 with the Air Force Systems Command Electronic Systems Division and Rome Air Development Center.)

2. B. Boehm, et al, Characteristics of Software Quality, North Holland Publishing Co., NY, 1978.


SOFTWARE DEVELOPMENT PROCESS
ITERATION DUE TO IDENTIFICATION OF PROBLEMS
REQUIREMENTS ANALYSIS
DESIGN
PROGRAMMING & CHECKOUT
TEST
OPERATIONS & MAINTENANCE
QUALITY METRICS APPLIED VIA CURRENT CONTROL MECHANISMS
REQ. REVIEW
PDR
CDR
V&ATS REVIEW
ACCEPTANCE TEST
PROJECT MANAGEMENT (CONTROL)
ONLY A SUBSET OF THE REVIEWS, STATUS REPORTS, AND DOCUMENTS PROVIDED DURING A SOFTWARE DEVELOPMENT ARE SHOWN.

LEGEND	
REQ REVIEW	REQUIREMENTS REVIEW
PDR	PRELIMINARY DESIGN REVIEW
CDR	CRITICAL DESIGN REVIEW
V&ATS REVIEW	VALIDATION AND ACCEPTANCE TEST SPECIFICATION REVIEW

Figure 2. Progressive Application of Software Metrics

3. M. Fagan, "Design and Code Inspections and Process Control in the Development of Programs," IBM TR 00.2763, June 1976.

4. M. Halstead, Elements of Software Science, Elsevier Computer Science Library, NY, 1977.

5. G. Myers, Reliable Software Through Composite Design, Petrocelli/Charter, 1975.

6. P. Richards, P. Chang, "Localization of Variables: A measure of Complexity," GE Technical Information Series, 76CIS07, December 1976.

7. G. Walters, J. McCall, "The Development of Metrics for Software R&M," 1978 Proceedings, Annual Reliability and Maintainability Conference, January 1978.

8. J. McCall, P. Richards, G. Walters, "Metrics for Software Quality Evaluation and Prediction," Proceedings of NASA/Goddard Second Summer Engineering Workshop, September 1977.

9. Contract F30602-78-C-0216 with the Air Force Systems Command Rome Air Development Center and U.S. Army Computer Systems Command, AIRMICS.

ADDRESS

James A. McCall
General Electric Company
450 Persian Drive
Sunnyvale, California 94086
(408) 734-4980

RELIABILITY EVALUATION AND MANAGEMENT FOR AN ENTIRE SOFTWARE LIFE CYCLE *

ISAO MIYAMOTO

NIPPON ELECTRIC COMPANY, LTD
Tokyo, JAPAN

Effective software reliability evaluation requires theories of software reliability which define and deal with software reliability quantitatively, technologies for reliability data measurement and data analysis, techniques to estimate or predict software reliability, and practical reliability evaluation methodologies which effectively reflect characteristics of software nature.

This paper assesses the extents to which these requirements are currently met, and introduces some approaches toward an effective software reliability evaluation. Introduced are the methodologies for software reliability evaluation and the software reliability management-aid tools.

1. INTRODUCTION

Since a software reliability became the most serious and important problem in software industries, we have already spent more than 15 years. During this period, many people tried to overcome this problem vigorously and various kinds of approaches were taken from all angles. Now, it is broadly recognized that a software reliability problem is clearly an integral portion of software engineering. However, as known well, inspite of all our efforts, the software reliability problem still remains as the knottiest subject. This may call for next severe question. What have we done for a long time, how on earth? Clearly, some portion of the reliability problem might be due to evaluation techniques for software reliability.

Practically, an effective software reliability evaluation requires theoretical basis. Theories of reliability should define software reliability and its metric quantitatively. Also needed are measurement and analysis techniques for reliability data represented in terms of the metric, techniques to estimate or predict future software reliability growth in testing stages, practical evaluation methodologies reflecting software characteristics effectively, and evaluation-aid tools.
From these points of view, this paper assesses previous works in software reliability evaluation briefly, extracts major problems, and tries to give them some solutions or improvements. Proposed ideas are integrated as a reliability management-aid system for an entire software life cycle.

Before we start a discussion, we must note some basic terms in order to avoid confusion. Software error is a defect that causes a software failure. Software failure is an unacceptable departure of software operation from software requirements. Software reliability is defined as a probability of failure-free operation in a specified environment for a specified time. (This definition will be changed later.) Reliability growth is defined as the increasing probability of software to perform required functions under stated conditions during stated time interval. Software reliability model mainly refers to a mathematical model constructed for a purpose of assessing the reliability of software from specified parameters which are either assumed known or are measured from observations or experiments on software.

2. SOME BASES FOR SOFTWARE RELIABILITY

Firstly, let us review previous works in software reliability evaluation.

We can easily find out two major directions in studies done by predecessors. Namely, they are 1) a direction to study reliability characteristics of software itself, and 2) a direction to study software reliability evaluation technique. Let us look into them in turn.

2.1 Nature of Software Reliability

The first direction of studies aims to find out fundamental elements of software which affect software reliability, and to make clear causal relationships between these reliability elements and software development technologies. In other words, this is to analyze the elements of software quality and data on when, where, how, and why people make software errors. Also intended as a final object of this direction is to establish a set of effective and reliable software development techniques and methodologies.

* THIS PAPER IS PARTIALLY PUBLISHED AT THE THIRD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, MAY 1978.

(1) Elements of software reliability

There are several works in analysis of software reliability elements, and most of them are by-products of general study for software quality which aims to evaluate software quality qualitatively or quantitatively. Organized software quality evaluation was firstly done by Rubey and Hartwick [1]. Brown and Lipow [2] formulated some number of quality metrics, and Wulf [3] identified and provided concise definitions on seven software quality attributes. Abernathy, et al, [4] defined a number of characteristics of operating systems and analyzed some tradeoffs between them. Recently, a framework for software quality characteristics was established by Boehm, Brown and Lipow [5]. They clearly identified the software reliability elements, in addition to a maintainability and a portability: namely, self-containedness, accuracy, completeness, robustness/integrity, and consistency. A large number of software quality-evaluation metrics were defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation.

However, for an effective and quantitative evaluation, we need more amount of further work in this field.

(2) Software reliability versus Hardware reliability

As a typical study of software reliability nature, there are some analyses discriminating natures between hardware reliability and software reliability [6,7, 8]. Some of fundamantal differences between the natures of software and hardware that affect reliability evaluation are:

a1) Software systems never run and software errors cannot be met without any input.
a2) Software is the transformation of designer's idea into a symbolic language for computer processing and software reliability is only dependent on correct design and the expression of this design. Hardware, in addition to having the reliability problems in correct design and expression of the design, is physical in nature and subject to component failure patterns that are statistically measurable.
a3) Software components do not degrade with time as a result of environmental stress or fatigue effect (i.e. wear-out).
a4) No imperfections or variations are introduced in making additional copies of a piece of software (except possibly for a class of easy-to-check copying errors).
a5) A correction of software fault alters configuration of software and eliminates any possibilities of its reoccurrence. Repairs of a software configuration tend to alter the configuration, unlike most hardware component replacement repairs.
a6) A comprehensive failure mode and effect analysis is impractical for large software, because of the large number of distinct logic paths.
a7) The information provided by detected errors has not yet been accurately characterized, limiting its usefulness in predicting the remaining number of errors.
a8) There is no standardized approach for exhaustively testing software in order to assure that it meets all operational requirements.

These affect severely to the software reliability evaluation techniques, especially to the modelling, and lead us to be careful in applying hardware reliability theory to software.

(3) Software error data collection and analysis

A main portion of the software reliability nature analysis might be a study for software error data collection and statistical analysis. Namely, that is to collect and analyze information on when, where, how and why people make software errors.

For several years past, many people have made much efforts to collect software error data during software development and maintenance processes. We find some published reports on software error data collections. Such data are, of course, necessary to evaluate a reliability of a software product itself, and also useful to derive and validate software reliability models. However, strictly speaking, very few data have been collected firmly through an entire software development and maintenance phases. Let us look all around and give some example studies on software error data.

Endres [9] collected error data discovered in testing stage of operating systems development, and analyzed them in terms of a distribution of error occurrence rate and frequency of software modification for each module. The report contains classified error statistics by error types and causes. Shooman and Bolsky [10] introduced statistical data on frequencies of error occurrence rate for each error type, computer time and working time used to test, computer time and working time to correct errors, changes for software, and some other with trouble and correction report forms. Thayer, Lipow and Nelson [11,12] analyzed error data collected by some projects in TRW precisely. Besides these, there are many interesting works that report various kinds of error data from various points of view: studies done by Boehm, et al [13], Bell and Thayer [14], Fagan [15], Akiyama

[16], Gannon and Horning [17], McGeachie[18], Musa [19], Litecky and Davis [20], Baker [21], Miyamoto [22], and Weinberg [23], for instance. However, most of the previous studies collected and analyzed only errors discovered within testing stages. Few exceptions are an analysis of software requirements errors by Bell and Thayer [14], and a design error analysis by Fagan [15]. Software error information should be collected and analyzed through entire software validations and operations of whole software life cycle.

Most of the previous works collected error information manually by using document forms. See references [10],[11] and [15] for instance. There is no systematic approach to collect error information automatically by using an automated data collection technique. Though SIMON [24,25] and Software Factory [26] are aiming to automate a collection of project management data, they only can collect software error data to be detected in testing, not for whole life cycle of software.

When we analyze software error data in testing and operational stages, we have to take into account following thing. Since software systems never run and software failures cannot be detected without any inputs, we can not find new errors without changing a test domain or a user domain[22]. Then, if we collected information on change of test domain or user domain in addition to error information at same time, we would be able to analyze more detailed insights on software reliability. This point of consideration is often omitted in most of previous works.

On the other hand, it is extremely difficult to compare error data from different sources. It is a big problem that the results of previous error data collection and analysis efforts are not compatible for each other. This might be mainly caused by a lack of standards and unified approaches on definitions of terminologies used in projects, error categorization, data collection procedures, data analysis, software error database, and etc. Though it is reported that some efforts are now under way at USAF Rome Air Development Center, and within IEEE Technical Committee on Software Engineering [7], formal outputs are not available by now. Because it requires much time and money to build a common error database, we have to collect and analyze error data in a compatible fashion as far as we can.

(4) Problems in error data collection and analysis

Based on the review and some references [9,10,11], we can summarize major problems in error data collection and analysis as follows:

b1) Software development and maintenance projects, the software, and the reliability data vary considerably and not describable in common terminology. It is extremely difficult to compare error data from different sources.

b2) Some projects produce data that are classified as they like.

b3) Analysis techniques and questions to ask of the data are not well known.

b4) Data accuracy is a chronic question.

b5) Analysis is often incomplete or inaccurate if proper communication with project performers is not established.

b6) Project organizational structure and resources vary, making consistent, multi-project data collection questionable.

b7) Definition of which parameters are needed and meaningful to collect is in its infancy.

b8) Analysis of relationships between error information and the system operations (i.e. test domain, user domain) is often left unnoticed.

b9) Definitions of measures to represent program complexity and scale are needed. Analysis of correlations between program complexity or scale, and the error information (e.g. frequency) is needed.

b10) Manual data collection is a lot of work.

b11) Certain data items are perishable and must be collected and analyzed timely when they become available, not after the fact.

b12) There is no guarantee that data will be collected (i.e. no requirement for projects to collect data).

b13) The fervor of data collection inspires data gathering that is non-supportive of software development processes.

b14) Presently implemented data collection schemes often fail to gather data in sufficient detail, making results of analysis questionable.

b15) Software error data collection is commonly done for testing stages, not for entire software life cycle.

b16) Software reliability data collection can represent cost, schedule, and manpower impediments to software development projects. The impact or cost considerations of data collection, although real, are not fully appreciated.

b17) Performers, project management, and even the customers of software are sensitive about providing data that might be used to adversely evaluate project by external agencies.

b18) Contractor and customer representative of project management are not aware of the benefits of data nalysis and therefore tend not to support it.

b19) Project structure is generally not tailored to use available data (i.e. the mechanism for analyzing data and folding results back into the project is not provided).

b20) Some data elements require protection to preserve the privacy of the contributor (e.g. cost data).

b21) Data collection is commonly thought to be "not necessary" to a properly managed project.

Some of these problems are due to the policy of software project management.

2.2 Software Reliability Evaluation

The second direction of studies is to study a phenomenological aspect of software reliability. This is to study how often a software is deficient while it operates, in other words. Also, this direction aims at gaining high quality of software as a final object. However, a greater emphasis is placed on to establish an effective software reliability evaluation technique itself.

(1) Metrics for software reliability

At a standpoint of phenomenological aspect, a software reliability and metrics to represent a degree of software reliability are often defined probabilistically as defined before. Metrics are defined sometimes time-dependently and sometimes time-independently. Most metrics were developed originally for hardware reliability: for instance, availability A(t), reliability function R(t), mean time to failure MTTF, mean time between failures MTBF, and mean time to repair MTTR. In addition to these, there are some other metrics, such like mean time between software errors MTBSE [22], some rates to represent reliabilities of input domain, test domain, and user domain [22,27]. However, these work at times, but often are unable to explain actual experienced software reliability phenomena. This is primarily because of fundamental differences between software phenomenology and the assumptions of hardware reliability theories, and secondly because of a lack of practical methodologies for application to software. There is, therefore, no single metric which reflects the software natures and can give a universally useful rating of software reliability. This might be thought as one of reasons for a traditional criticism insisting that it is impossible to apply hardware reliability theories to software reliability problem.

(2) Software reliability models

Now let us review a software reliability modelling which is the most theoretical portion of software reliability evaluation techniques.

Shooman [28,29], Jelinski and Moranda [30,31] proposed similar type of probabilistic models for a removal rate of software errors during test. For these earliest software reliability models, they assumed an error detection rate was proportional to a number of remaining errors. Shooman related an error detection rate also to a program size and an instruction processing rate. He investigated some other types of models for error correction, and proposed a two-point parameter estimation approach. Schneidewind [32] suggested an empirical reliability prediction model by fitting failure intervals with an appropriate reliability function. He applied a maximum likelihood estimation to determine parameters of error detection and correction processes. Littlewood and Verrall [33] proposed a Bayesian reliability growth model by assuming that error corrections make smaller a failure rate in a probabilistic rather than deterministic fashion. Trivedi and Shooman [34] developed a many-state Markov model for an estimation of reliability function R(t) and availability A(t) during testing stages. They assumed that error and correction occur alternately and sequentially, only one type of error occurs, and error discovery rate is constant, in addition to basic Markov assumptions. Musa [6] developed a set of execution time model and calendar time model. He assumes that tests are representative of the environment in which a program will be used and are continuously global, failures are independent of each other and distributed at any time with a constant average execution time occurrence rate that is proportional to a number of errors remaining, all failures are observed, and an error correction rate is proportional to a failure detection rate at all time, for the execution time model. Also, in order to relate a calendar time with testing activities, he made some assumptions for utilization of resources (i.e. failure identification personnel, failure correction personnel, computer time). The model parameters are calculated by using a maximum likelihood estimation. The effectiveness of models has been validated at several real projects [35]. This is one of few models which have a practical methodology and a tool for application. In addition to these models, there were some other models developed by Schick and Wolverton [36], Weiss, Corcoran, and Nelson [37]. These are well reviewed by [11] and [48]. Also, Littlewood [38,39], and Shooman [40,41] have proposed some revised versions of their reliability models vigorously. The models proposed by Littlewood are based on Markov processes. Shooman has developed structural model in which logical paths of program structure, execution time and failure rate of each path are related with.

Besides, there are some other simple models to estimate a number of initial latent errors in programs. Such models do not assume much, and some models are based on the statistics of historical data [43]. For instance, Mills [42] developed an error seeding model. These models are useful sometimes.

(3) Problems in reliability models

During the review, we found many problems in software reliability models for practical applications. Most of them are derived from assumptions applied in the modelling [8]. Typical worrisome (?) assumptions are summarized as follows:

c1) The errors remaining in program decrease monotonically.
c2) The discovery rate is proportional to the number of remaining errors.
c3) New errors are not introduced during debugging.
c4) The software failure rate (and associated hazard rate) is constant, or increases, or decreases during failure intervals.
c5) The hazard rate increases or decreases at each time when error is detected.
c6) Software errors have the same likelihood of detection.
c7) All failures are observed.
c8) Data inputs to software system are randomly selected.
c9) Software failures are independent.
c10) Software errors have the same effects on system operations.
c11) Test and users operations cover the entire input domain of software and do not change.
c12) Error correction rate is proportional to failure detection rate.

Some of the reliability estimation problems are originated from assumptions of failure distribution, and some others are originated from assumptions of software operational profiles in testing stage and operational stage at users. In most models, it is assumed that a test domain and a user domain are fixed and do not change. We should note that a software reliability is a function of, not only a number of remaining errors, but also their effects on system operation, software operational profiles (which are changeable according to the test space growth and the usage patterns of software functions), and locations of remaining errors. In other words, the operational reliability is a probability that users don't enter specific inputs which inevitably encounter the latent errors.
Surprisingly very few practical model application methodologies for the reliability test data measurement (which should be separated from testing and debugging) and compilation are prepared in the existing software reliability models.
In a theoretical sense, there is no single model which can give a universally useful estimation of software reliability. And still, we are a long off from having truely reliable software reliability estimation models.

As a whole, we can conclude that we have very poor software reliability evaluation techniques and error database, for the present.

3. SOFTWARE RELIABILITY MANAGEMENT SYSTEM

After considerable study of previous works and experiences of actual projects, we have decided to improve the effectiveness of current software reliability evaluation by developing the application methodologies of currently existing techniques and models, and supporting tools. (We have not make plans to develop new models and techniques, although we have no powerful ones.)

For the problems of reliability evaluation, we shall deal with mainly by using multiple reliability metrics at the same time, preparing specific reliability data measurement method, relating with operational profiles, categorizing failures according to their effects on system operations, and limiting the objects of reliability estimation. The effectiveness of error data analysis can be basically improved by developing automated tools and methodology for data collection and analysis, and by integrating them with an automated system which supports entire software development and maintenance processes. By using such a system, as we continue to collect and analyze more and more data on how, when, how often, where, and why people make software errors, and how people detect and correct software errors, we will be able to get clear insights on how to evaluate software reliability, how to develop practical models for prediction, how to avoid making software errors, and how to organize the validation strategies.

Along this baseline, we are designing an automated software reliability management tool, and a highly ambitious software development and maintenance support system. This section will introduce a brief description for a concept of this system.

3.1 Software Development and Maintenance Support System

In order to support various activities for large scale software development and maintenance, SDMSS (Software Development and Maintenance Support System) is designed to have several basic subsystems as shown in Fig. 1. Requirements Engineering Subsystem consists of a language processor for a requirements definition language RDL, and analysis tools which examine correctness, consistency, completeness, and feasibility of software requirements described in terms of RDL. Requirements descriptions are stored into a development database and maintained as an abstract system model. Design Subsystem consists of a processor for software design language SDL and tools to examine correctness, consistency, completeness and efficiency of designed software.

Design descriptions are stored into database and maintained as a logical system model. Programming Subsystem consists of programming language processors and source code analysis tools. Static code analysis is done by program structure checker, module interface checker, events sequence checker and diagnostic functions of language processors. Test Subsystem is made up of test description language processor, and tools to support test case selection, test data generation, test execution, test result reporting, maintenance of test database for dynamic tests. A test description language TDL is useful to describe test drivers and stubs. Maintenance Subsystem supports maintenance activities originated from software problem reports or maintenance orders. History of software modifications is recorded in maintenance database. This subsystem internally uses functions of Requirements Engineering Subsystem, Design Subsystem, Programming Subsystem and Test Subsystem to redefine software requirements, redesign, recode, and retest. A tool to find out modules, specifications and test cases associated with a requested correction or modification is provided. In addition to these, SDMSS has Product Management Subsystem, Document Subsystem, Project Management Subsystem, and Software Error Management Subsystem (SEMS). Project Management Subsystem collects data on cost and project progress, and monitors software development and maintenance activities for project management.

Except Requirements Engineering Subsystem, SDMSS is now in design stage.

3.2 Software Error Management Subsystem

Associating with software validation activities and maintenance activities for an entire software life cycle, with a help by SDMSS, SEMS provides a set of methodologies and tools which mainly support a collection of software error data produced, a statistical error data analysis, a reporting error information to managers (e.g. QA manager) for swift feedbacking or feedforwarding to software activities, a management of error correction activities, and a prediction of software reliability during testing stages. SEMS intends to give solutions or improvements to some problems in software reliability evaluation and stimulate the rest.

(1) Components

SEMS consists of four major components. Error Management Program manages and maintains error information registered and status information of correcting activities, and controls interfaces to other subsystems in order to collect error data automatically. Reliability Estimation Program calculates a future software reliability at testing stages. This program contains two different reliability models. The Reporting Tools output statistical reports on categorized errors, and status reports on error correcting activities periodically and at any time when needed. SEMS is extensible to add new tools. Error Database is made up of events file, relation file, and statistic file as shown in Fig. 2. Events file maintains error information in chronological order of occurrence. Each error occurrence uses one record. Example error record format for dynamic testing stage is also shown in Fig. 2. An error record contains four parts: the first part keeps, in main, initial information on a situation of failure detection, the second part maintains status information for correction activities, the third part keeps formal results of error cause analysis, and the last part is for comments. Relation file maintains relations between error records. Relations indicate sets of categorized errors in terms of, for example, requirements errors, design errors, coding errors, or for each version of software, module, and many others. Reporting tools can make reports very easily by refering to this file. Statistic file contains various kinds of statistics on error data, such as, accumulated number of error occurrences, accumulated number of corrected errors, current number of errors remained uncorrected, and reliability estimation results.

Totally, SEMS has 6 basic types of procedures to interact with these components. They are defined for a manual registration of error information, an automatic registration of error information, a status registration of error correction activities, an inquiry for statistical error information or status information of each error, an authorization of error, and a reliability estimation.

(2) Sources of error information

Since, in general, software projects have a potential for creating a tremendous amount of various kinds of data, and error data are largely a by-product of software development and maintenance processes, we need to collect meaningful data items efficiently and timely for an entire software life cycle. Fig. 3 illustrates a typical software life cycle by phase and several types of software error data. These software error data form a continuum of requirements errors, design errors, and coding errors starting as early as the software requirements definition phase and extending into the operational phase. Software validations as error information sources by SDMSS are as follows. In requirements definition phase, some of software requirements errors can be detected by RDL processor and associated static analyzers automatically, and some others can be detected by an inspection review of requirements specification manually. In design phase, some of software design errors can be detected by SDL processor and associated tools automatically, and some others can be detected by an inspection review of design specifications manually. Errors on software efficiency can also be detected by using a simulation tool. In coding stage, some types of software errors are detected by diagnostic functions of language processors and static source code analyzers automatically, and some others are detected by an inspection review manually. In testing stage, the rests of software errors should be detected by dynamic tests. Since results of tests described by using TDL are self-checkable, failures are detected by automated tool. In other cases, failures are detected by manual analysis and reported in terms of software problem reports. In operational phase, users find software failures and report them manually by using software problem report. All of these activities generate error information.

There are three types of information sources. The first one is a static test result of specifications or source codes. This kind of error information can be sent to SEMS automatically by static analysis tools. The second type is a dynamic test result. In this case, Test Management Program in Test Subsystem sends error information to SEMS automatically. The third type is a result of manual inspections. This kind of error information must be registered to SEMS manually. Generally, a manual technique is easy to implement, but it requires manpower committed to a collection task. Automated techniques ease a pain of collection task, and can collect error data effectively and timely when it is generated. However, automated techniques tend to be less flexible in response to changes in software project demands and may cause cost problems of implementation. And yet, it is often needed to judge manually in order to collect meaningful data. Then, SEMS has employed both of manual and automatic data collection techniques so that meaningful error data can be collected effectively.

(3) Supporting functions for automated data collection

Now, taking a case of a test description language TDL, we would like to introduce supporting functions for an automatic data collection. Since a complete description of TDL is beyond the scope of this paper, we introduce only some features of TDL and *TDL processor for a test driver* description. (TDL is partly based on [46].)

A TDL description for a test driver has a following general form.

```
    <test description >
    <test definition- 1>
    <test definition- 2>
           .
           :
    <test definition- n>
     END
```

Each test defined in the test procedure description causes a partial execution of target modules. Tests are performed one at a time in the order in which they appear in test definitions. In the test description, what a set of tests will examine is written with information on testing environment. A complete form is as follows.

```
<test description> =
     <test case identification>
     <error category identification>
     <hardware configuration identification>
     <software configuration identification>
     <test system identification>
```

These information specified in test description are sent to SEMS at time when a failure is met. Each test execution of target modules is described in a test definition. A test definition has four items as shown in next.

```
<test definition- i>=
     <initialization codes>
     <assertions>
     <execution directive>
     <error category identification detail>
```

Immediately before a test execution, the variables specified in the initialization code statements are initialized. A form for initialization is

```
<target module name/variable,initial value>
```

The most test definitions contain assertions about the performance of the target modules. They are specified in VERIFY statements. A form is

```
 VERIFY [at L1,L2,..,Li,..,Ln](assertions)
```

Li is a label reference to target modules, where the assertions are to be verified, and is written in a form [module name:label]. When no labels are specified, the assertions are verified immediately after the test execution terminates. An assertion can be specified in any way of next three types:

```
 (module name: logical expression)
 (PATH-IS: regular expression)
 (TIME-IS-LESS-THAN: value)
```

Logical expressions are test predicates using variables and values. An argument to a PATH-IS assertion is a regular expression made up from operation names. Regular expressions permits use of four operators. The operator ; is a sequencing operator, specifying an order in which the modules are supposed to be executed. The operator + is an alteration operator specifying either one of the modules can be executed. The operand * is used to denote repetition. The operand

& can be used to denote concurrent processing of modules. The assertion for TIME-IS-LESS-THAN is used to check a execution time of target modules. Every test definition is executed by an EXECUTE statement of the following form.

```
EXECUTE [FROM L1][TO L2,...,Li,..Ln]
```

where Li is a label reference to target modules. This statement specifies the first and last statements to be executed. A STOP or RETURN in target modules terminates only the current test execution. The error category identification is a detailed category identification for an error which is expected to be detected by this test definition, and should be consistent with senior category identification specified in the part of test description.

Internal procedures for an automatic collection of error information is roughly shown in Fig. 4. Firstly, immediately after a test execution, a result is checked whether it agrees with a specified assertion. In the case that it does not agree with, an event indicating a failure occurrence is informed to SEMS, and the information written in a part of test description and error category information are entered into Error Data Queue by test management program. At this time, test management program adds a calendar time when failure was detected, name of inspector for current test execution (Test Subsystem knows a name of current user), and error identification number which is serial through the current validation activity. On the other hand, SEMS takes out an entry of error information from the error data queue and stores it into error database one by one. At this time, a serial number which is counted through whole software life cycle is added to the second part of an error identification number. At a time when a failure event is registered in error database, since an error has not been judged formally as a true software error, this is recognized as a quasi error temporarily. A true cause and a location of the error can not be made clear until all failure analyses complete, ordinarily. Formal analysis results often do not agree with an initial judgement for a quasi error. The formal error information are stored in the third part of error record by QA manager in manual. Besides, as correction activities progress step by step, each status is registered and updated with information on committed working hours, computer time used, and some others in the second part of error record. These information are useful to manage error correction activities.

3.3 Software reliability estimation

SEMS provides a software reliability estimation capability and prepares a methodology for an estimation models application.

(1) Software reliability models

Reliability Estimation Program contains two different software reliability models: they are a model developed by Musa [6] (The calculation modules for this model in Reliability Estimation Program is originally developed by Musa [47].) and a simple model developed by the author [22]. As reviewed before, Musa's model is based on several assumptions, and the author's model is also based on some basic assumptions. These are not exceptions. However, we should like to bring up basic questions here. (In this section, we discuss reliability models from a viewpoint of practical use, not a theoretical aspect described before.)
Are these assumptions (e.g. assumptions c2, c4, c7, c8, c9, c11, and c12) realy quite inappropriate?
We are sure that they are not the ideal models. By satisfying some conditions in some ways, can't we use them for a restricted environment specifically?

Answers might be probably proved by our experience and intuition. If we could have some specific methodologies for a measurement and compilation of reliability test data, restrict objects of application, and modify debugging and testing processes in order to satisfy the assumptions, we would be able to decrease their bad effects on the validity of estimation. As a matter of fact, the effectiveness of Musa's model has been validated by several real projects[35]. The author's model is developed only for a highly dedicated online real time system and has specific reliability test data measurement and compilation methods [22]. We may say that we would be able to apply them for practical use if only we had taken some points into account. In spite of this possibility, some theorists might say that, in any case, it is still difficult to estimate universally by a single model.
There is a good answer to this. In order to obtain more effectiveness, it is better to apply multiple models to software reliability estimation jointly. This is a main reason why SEMS has two different models. The author's model is now under way for the revision. Revised model is going to reflect the changes in system operational profiles during testing and operational stages.

(2) Methodology for application

A procedure to estimate software reliability in SEMS environment is shown in Fig. 5. Input data to reliability estimation program consists of a number of parameters which are grouped into 5 categories: failure, planned, debug environment, test environment, and program. The failure data consists of a set of execution time intervals between failures, along with a number of days from the start of testing on which failures occurred. In a category of planned data, there are available computer time (measured in terms of prescribed work periods), number of available failure correction personnel, number of available failure identification personnel, computer time utilization factor, failure correction personnel utilization factor, and objective mean time to failure. In a category of debug environment, there are average computer time expended per unit execution time, average computer time required per failure, average failure identification work expended per unit execution time, average failure correction work required per failure, and the average failure identification work required per failure. A testing compression factor is a parameter of test environment. As parameters of category of program, there are a number of failures required to expose and remove all errors Mo, and initial MTTF at start of testing. The parameter Mo must initially be estimated from a number of inherent errors and an error reduction factor by using previous statistics [9,11, 16,22] or by using simple models [43]. However, once data are available on intervals between failures during test or operation, these parameters may be reestimated. These data can be provided by reliability test data measurement, error database, and project database. Ways to handle these parameters are well described in [35].

Reliability test data measurements are different from ordinary testing activities and are done periodically or at any time needed in order to collect data on current reliability. (Test cases are generally chosen to quickly detect errors and are not representative of operational use.) During reliability test data measurement, execution time intervals between failures are measured for each categorized class of failures. Failures are categorized according to their effects on system operations. Example categorization gives four classes of failures: catastrophic failure which leads to system down, serious failure which causes malfunctions on all current users, moderate failure which causes malfunction on one of current users, and trivial failure which is typographical error in output messages. Since an error correction may alter a software configuration and decreases statistical meaning, detected errors are not corrected for the currently used system during current data measurement. Input data in test space are selected through a prediction of user space initially, and are not changed through the measurement. Test space must be reflecting possible combinations of basic users operational patterns. (Recall that software reliability is affected by system operation strongly.) Sample reliability test data measurement methods are introduced in [22] and [28].

Using collected data, two models estimate the software reliability growth. Estimation might be plotted as shown in Fig. 6 with measured reliability, for instance. It is strongly recommended to evaluate software reliability for each categorized class of failures. A priority of failure analysis and correction activity, and an allotment of resources for each category of failures must be determined prior to the estimation.

The software systems as the best objects of reliability estimation by SEMS should have following characteristics. Namely, their input spaces must be large enough, and used randomly and independently by many number of users simultaneously (for random input). A unit of system services (e.g. an interaction or a connection) must be small, and system operational profiles at users should be predictable and stationary at any time while they operate. System operations can be classified into some states clearly in order to ease an effect analysis of software errors on system operation.
From these points of view, SEMS recommends to apply its estimation technique to online real time transaction-driven systems with a number of user terminals, in main, such like large bank systems, reservation systems, inventory systems, information retrieval system, electronic switching system, and etc. Typical example system is shown in [22]. In the case of other type of systems (e.g. batch processing systems), effectiveness of estimation might be decreased more than above cases.

Finally, we should note that estimated results do not show an absolute certainty in the future reliability growth. They are only useful to fix our aim. Because software reliability growth rates are often changed by a project management policy, ways of testing and debugging, individual abilities, positivisms and morals of project performers, and by many other factors greatly.

4. CONCLUDING REMARKS

We have reviewed the software reliability evaluation techniques, and described some approaches to improve some of the major problems, briefly. Finally, let us assess the effectiveness of SEMS on the software reliability evaluation, especially on what will be improved by SEMS.

(1) On error data collection and analysis

Basically, the automatic data collecting function of SEMS which operates with SDMSS as one system may improve the problems of b10, b11, b12, b13, b14, and b15. For the rest, we may classify them into three categories: 1) the problems on standardization or unification (i.e. b1, b2, b3, b4, b5, b6, b7, b8, and b9), 2) the problems associated with the project management policies (i.e. b17, b18, b19, b20, and b21), and 3) the cost problem (i.e. b16).

The first kind of problems may be improved by piling up the use experiences of SDMSS. Since SDMSS and SEMS provide the standardized methodology, terminology and techniques for the software, software development and maintenance project, and software reliability evaluation, the pile of collected and analyzed error information would be compatible for each information source. In order to solve this kind of problems in the broad sense, we may need some theoretical works and the joint efforts with public organizations. The poor compatibility of error data is caused partially by the lack of the valid measures to represent program complexity and scale. The statistical error data analysis must be done in terms of such measures independently with the kind of programs and programming languages or design languages. The theoretical work is needed to get the valid and common measures for software. Currently there are some statistical approaches to this. For instance, Akiyama [16], Thayer, Lipow and Nelson [11], Halstead [44], and Green [45] have reported some results on the correlations between error data and this kind of measures. In SEMS, since the relations between error database and the development database are maintained, it is rather easy to analyze the correlations between program complexity and scale, and error data statistics. (c.f. b1, and b9) Besides, SEMS provides a set of error categorizations. They may categorize errors according to their causes, their effects on the system operations, their locations, the phases in which they are introduced, their hardness for correction, the phases in which they are detected, the detection methods, and other characteristics. In the environment where SEMS is used, since the multiple categorizations are made, the more amount of information is provided than the cases that single categorization is employed. (c.f. b1, b2, and b6)

Most of statistical works and reporting works are done by SEMS. Thus, users are not needed to do troublesome work. SEMS is extensible to add new tools by users in order of extraction and analysis of error information. (c.f. b3) The data accuracy may be improved and balanced more than before, because SEMS collects error data in standardized manner. (c.f. b4) On the other hand, since SEMS is jointly used with SDMSS in general, the analysis works can be done on intimate relations with the development and maintenance activities. (c.f. b5) However, QA manager must take more responsibilities than before. SEMS can refer the information of system operations related with the error information in testing and operational stages. (c.f. b8) Generally, it is not well known that what kind of data items are meaningful and how much we need for software reliability evaluation during the development. Thus, it is important and meaningful to accumulate the error information and to gain the experience by using automated tool such like SEMS. (c.f. b7)

Most of the second class of problems are beyond the scope of SEMS. For instance, the problems b18 and b21 can not be solved without changing the consciousness of project managers. However, SEMS provides the protection function for error database. (c.f. b17, b20)

Finally, to implement and use SEMS do cost more than manual evaluation. Compared with the SDMSS, SEMS spends little amount of computer time and other resources. The cost problem is very common to every automated tool. Since the software reliability is the most important problem in software industries, we must do something for it. SEMS can collect and analyze error data more efficiently than other existing methods.

Totally, SEMS can improve some of the major problems on software error data collection and analysis.

(2) On software reliability evaluation

After the review of existing techniques, we have concluded that there is no single metric or reliability model which can give a universally useful reliability evaluation or an estimation, from the theoretical point of view. However, by taking into account the application methodologies which cover the measurement and compilation of reliability test data, restrict the object of application, and modify the debugging and testing processes to meet with assumptions made, this conclusion may be changed. In some specific environment, we can obtain useful estimation to fix our aim, by using multiple reliability models. Though we are developing new models vigorously, we should place greater emphasis on the development of application methodologies and the estimation-aid tools. In this sense, SEMS may stimulate the studies.

Finally, we would like to claim again. By using such a system we proposed, as we continue to collect and analyze more and more data on how, how often, when, where, and why people make software errors, and how people detect and correct software errors, we will be able to get clear insights on how to evaluate software reliability, how to develop practical models for predicting software reliability, how to avoid making software errors, and, how to organize validation strategies.

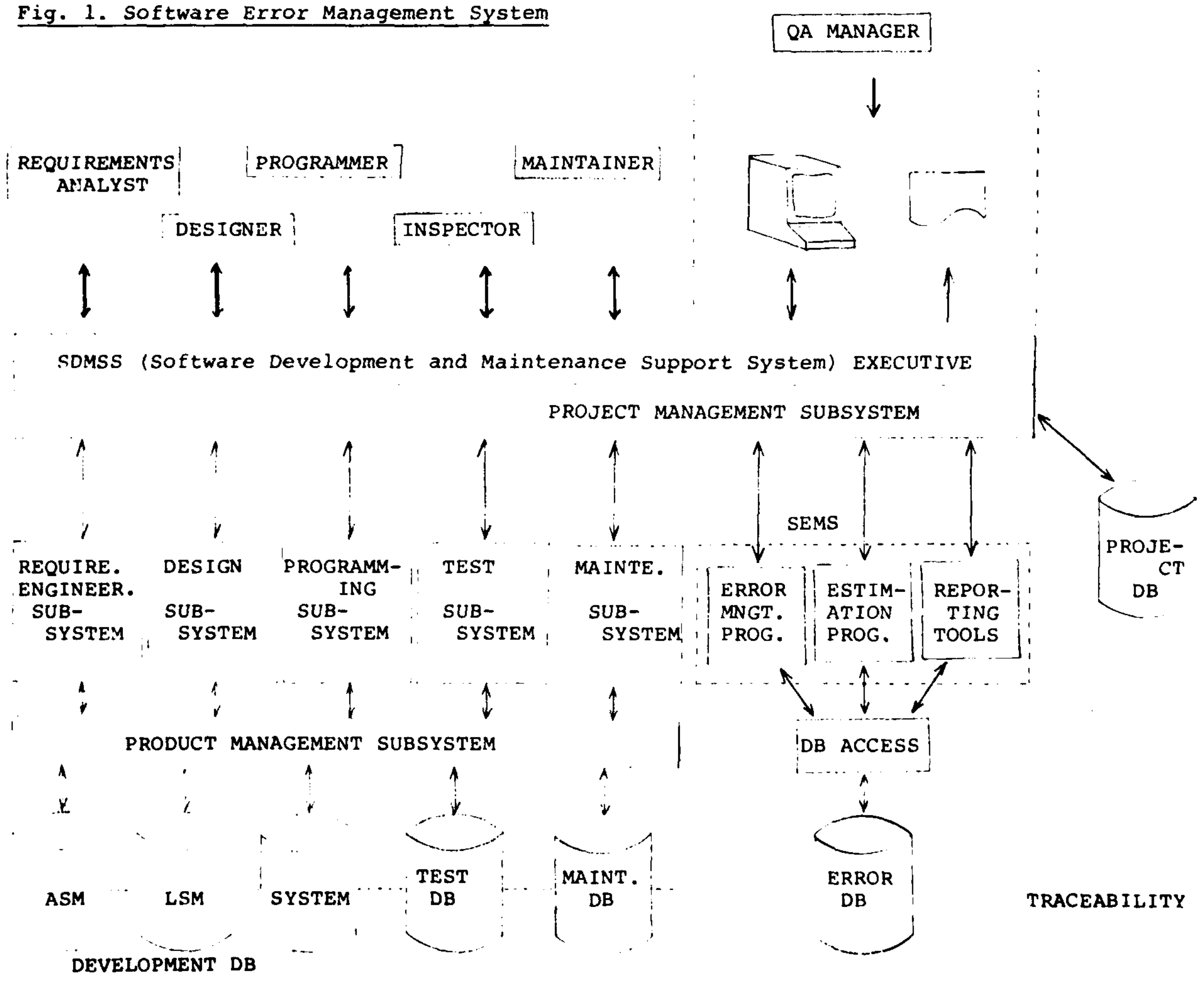
5. ACKNOWLEDGEMENT

Especial thanks go to John Musa of Bell Labo. for his valuable information and estimation-aid program for his model. Also, the author would like to thank Eldred Nelson of TRW for his helpful information.

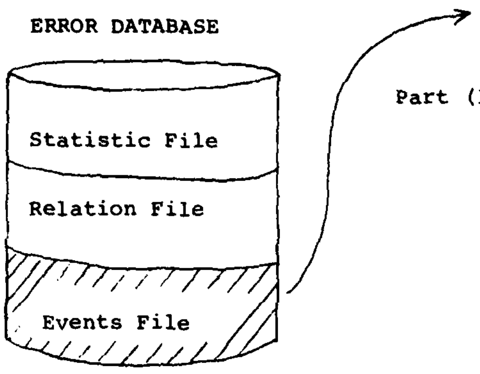
[1] R.J. Rubey and R.D. Hartwick,"Quantitative measurement of program quality", Proc. ACM National Conference,1968,pp671-677
[2] J.R. Brown and M. Lipow,"The quantitative measurement of software safety and reliability",TRW Report SDP-1776,1973
[3] W.A. Wulf,"Programming methodology", Proc. Symp.on High Cost of Software, Sept. 1973
[4] D.H. Abernathy,et al,"Survey of design goals for operating systems",GIT Report, GTIS-72-04
[5] B.W. Boehm,et al,"Quantitative evaluation of software quality",Proc.2nd ICSE, Oct. 1976, pp592-605
[6] J.D. Musa,"A theory of software reliability and its application",IEEE Tr.SE, Vol.SE-1,No.3,1975,pp312-327
[7] B.W. Boehm,"Software Engineering",IEEE Tr.Computers,Vol.C-25,No.12,Dec.1976, pp1226-1241
[8] R.A. Pikul and R.T.Wojcik,"Software effectiveness:A reliability growth approach",Proc.MRI Symp.Comp.Soft.Eng.,Apr. 1976,pp531-546
[9] A. Endres,"An analysis of errors and their causes in system programs",Proc. ICRS, April 1975, pp327-336
[10] M.L. Shooman and M.I. Bolsky,"Types, distribution and test and correction times for programming errors",Proc.ICRS, April 1975, pp347-357
[11] T.A. Thayer,et al,"Software reliability study", TRW System Rep.SS-76-03,1976
[12] T.A. Thayer,"Understanding software through analysis of empirical data",Proc. NCC 1975, pp335-341
[13] B.W. Boehm,et al,"Structured programming:A quantitative assessment",IEEE Comp. June 1975, pp38-54
[14] T.E. Bell and T.A. Thayer,"Software requirements:Are they really a problem?", Proc.2nd ICSE,Oct.1976, pp61-68
[15] M.E. Fagan,"Design and code inspections to reduce errors in program development", IBM Sys.J. No.3,1976,pp182-211
[16] F.Akiyama,"An example of software system debugging", Proc.IFIP 1971,pp359
[17] J.D. Gannon and J.J. Horning,"The impact of language design on the production of reliable software", Proc.ICRS,Apr.1975 pp10-22
[18] J.S. McGeachie,"Reliability of the Dartmouth Time Sharing System",IEEE Sym. Soft.Reliability,1973,pp117-123
[19] J.D. Musa,"An exploratory experiment with 'foreign' debugging of programs", MRI Symp.Comp.Soft.Eng.,Apr.1976, pp499-512
[20] C.R. Litecky and G.B. Davis,"A study of errors, error-proneness, and error diagnosis in COBOL",CACM Jan.1976,Vol.19, No.1, pp33-37
[21] F.T. Baker,"System quality through structured programming",Proc.FJCC 1972 pp339-343
[22] I.Miyamoto,"Software reliability in on line real time environment",Proc.ICRS, April 1975, pp194-203
[23] G.M. Weinberg,"The psychology of computer programming",Van Nostrand Reinhold, New York,1971
[24] R.J. Fleischer and R.W. Spitler,"SIMON -A project management system for software development",MITRE MTP-169,Apr 1976
[25] J.A. Clapp and J.E. Sullivan,"Automated monitoring of software quality",Proc. NCC 1974,pp337-341
[26] H. Bratman and T. Court,"Software Factory",Computer,May 1975,pp28-37
[27] W.H. MacWilliams,"Reliability of large real-time control software systems",Proc Symp.Soft.Reliability,1973,pp1-6
[28] M.L. Shooman,"Operational testing and software reliability estimation during program development",IEEE Symp.Comp.Soft. Reliability,1973,pp51-57
[29] M.L. Shooman,"Probabilistic models for software reliability prediction",Statistical Computer Performance Evaluation, W.Freiberger,ed.,N.Y. Academic Press, 1972, pp485-502
[30] Z.Jelinski and P.B. Moranda,"Software reliability research",same source as 29
[31] Z.Jelinski and P.B.Moranda,"Applications of a probability-based model to a code reading experiment",Proc.Symp.Comp. Soft. Reliability,1973,pp78
[32] N.F.Schneidewind,"An approach to software reliability prediction and quality control,"Proc.FJCC 1972,pp837-847
[33] B.Littlewood and J.L.Verrall,"A Bayesian reliability growth model for computer software",IEEE Symp.Comp.Soft.Rel. April 1973, pp70-77
[34] A.K.Trivedi and M.L.Shooman,"A many-state Markov model for the estimation and prediction of computer software performance parameters",Proc.ICRS,Apr. 1975,pp208-220
[35] J.D.Musa,"Measuring software reliability",Proc.National Meeting,ORSA/TIMS, May 1977
[36] G.J.Schick and R.W.Wolverton,"Assessment of software reliability", 11th Ann. Meeting, GermanOR Society,Sept. 1972
[37] E.C.Nelson,"A statistical basis for software reliability assessment",TRW Systems Report,March 1973
[38] B.Littlewood,"A reliability model for Markov structured software",Proc.ICRS, Apr. 1975,pp204-207
[39] B.Littlewood,"A Semi-Markov model for software reliability with failure costs", MRI Symp.Comp.Soft.Eng. 1976,pp281-300
[40] M.L.Shooman,et al,"Effect of manpower deployment and bug generation of software error models",ibid, pp155-170
[41] M.L.Shooman,"Structural models for software reliability prediction",Proc. 2nd ICSE,Oct.1976,pp268-280
[42] H.D.Mills,"On the statistical validation of computer programs", FSC-72-6015, IBM FSD, 1972
[43] G.J.Myers,"Software reliability: Principles and practices",John Wiley & Sons, 1976
[44] M.H.Halstead,"Elements of software science",Elsevier Comp.Sci.Lib.,1977

[45] T.F.Green, et al,"Program structures, complexity and error characteristics", MRI Symp.Comp.Soft.Eng., Apr.1976 pp139-154
[46] D.J.Panzl,"Test procedures: A new approach to software verification",Proc. 2nd Int.Conf.Soft.Eng.,Oct.1976,pp477-485
[47] J.D.Musa,"Program for software reliability and system test schedule estimation--User's Guide",IEEE Computer Society Repository, No.R77-244
J.D.Musa and P.A.Hamilton,"Program for software reliability and system test schedule estimation--Program Documentation",IEEE Computer Sciety Repository, No.R77-243
[48] G.J.Schick and R.W.Wolverton,"An analysis of competing software reliability models",IEEE Tr.SW.Eng., Vol.SE-4,No.2 March 1978, pp104-120

Fig. 1. Software Error Management System



ERROR DATABASE



Basic Record Format

```
Part (1)  .Error Id Number    #xxxxxx-xxxxxx
          .Category Id (tentative)
                 -classification #1,#2,#3,#4......
          .Detected by (name)
          .Date & Time detected
          .System Information
                 -Hardware Configuration Id
                 -Software Configuration Id
                 -Database Id
                 -Test System Id
          .Failure Location (tentative)
                 -Module(name,version & rev. #,author)
                 -Spec.(name,version & rev.#, page)
          .Associated Test Cases
                 -Validation Id
                 -Test Case and Data Id
          .Associated Trouble Report or
                 Maintenance Order Id.

Part (2)  .Status Information
                 -Nothing done (date)
                 -Error analysis (date/working hours)
          .Correction Started (date)
                 -Redefinition of requirements
                 validation          )
                 -Redesign           |data
                 validation          |computer time
                 -Recoding           |working hours
                 validation          )
          .Correction completed (date/work hour)
          .Demonstrative Test (date/work hour)
          .Find-and-fix cycle of error
          .Approved date

Part (3)  .Formal Judgement
                 -hardware error
                 -operation error
                 -testing error
                 -unknown (reoccurable?)
          .Software error
                 -formal category Ids #1,#2,#3,...#n
                 -formal location
                    module,page,line,author
                    spec.,page,line,author
                 -error produced by erroneous debug
                    original error id
          .Associated Errors (ids)
          .Accountable or not

Part (4)  .comments on this error
```

Fig.2 Error Database

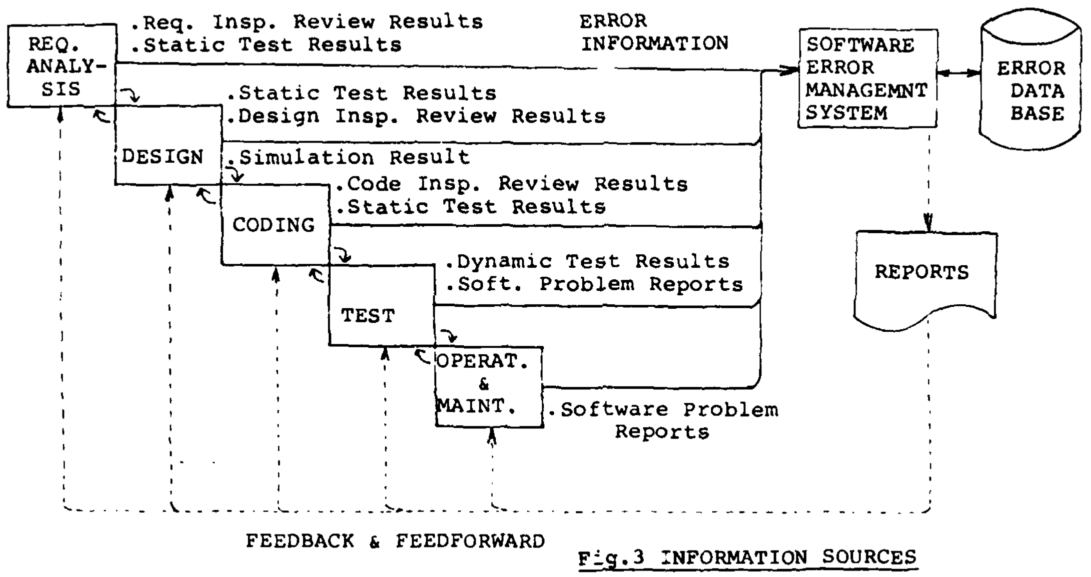


Fig.3 INFORMATION SOURCES

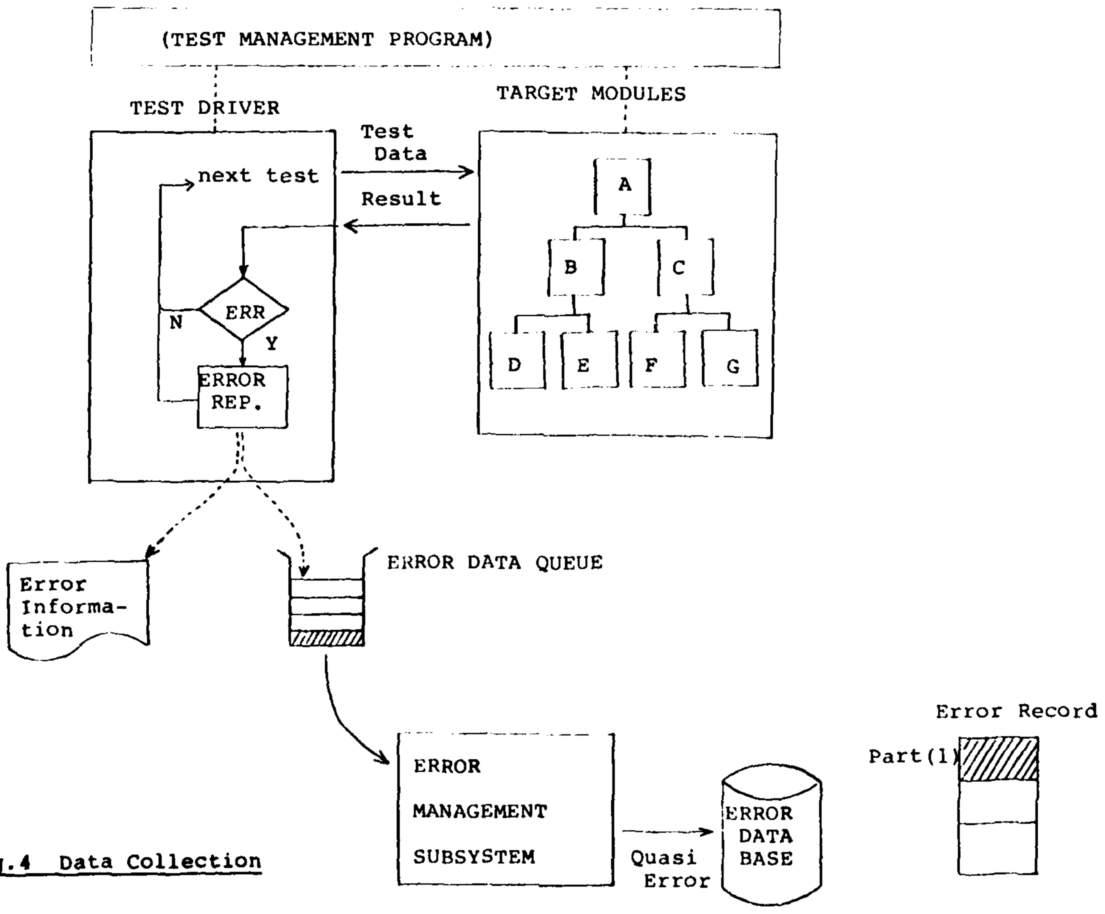


Fig.4 Data Collection

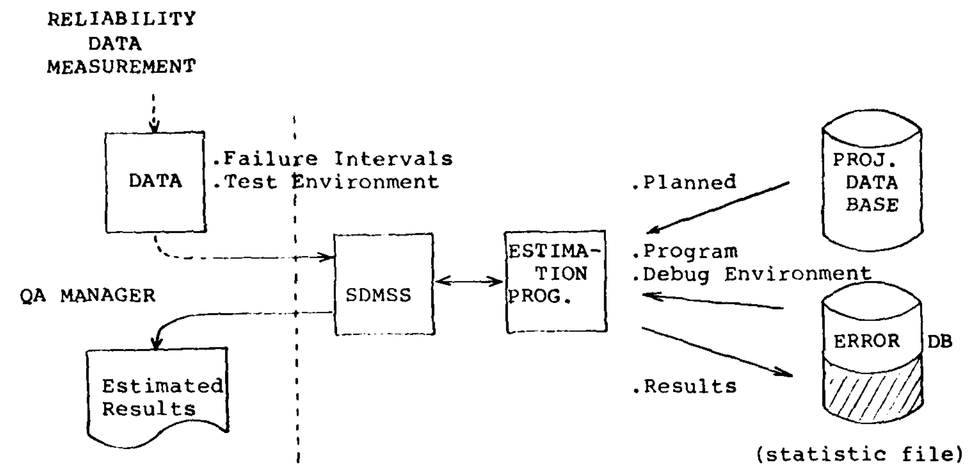


Fig. 5. Reliability Estimation Procedure

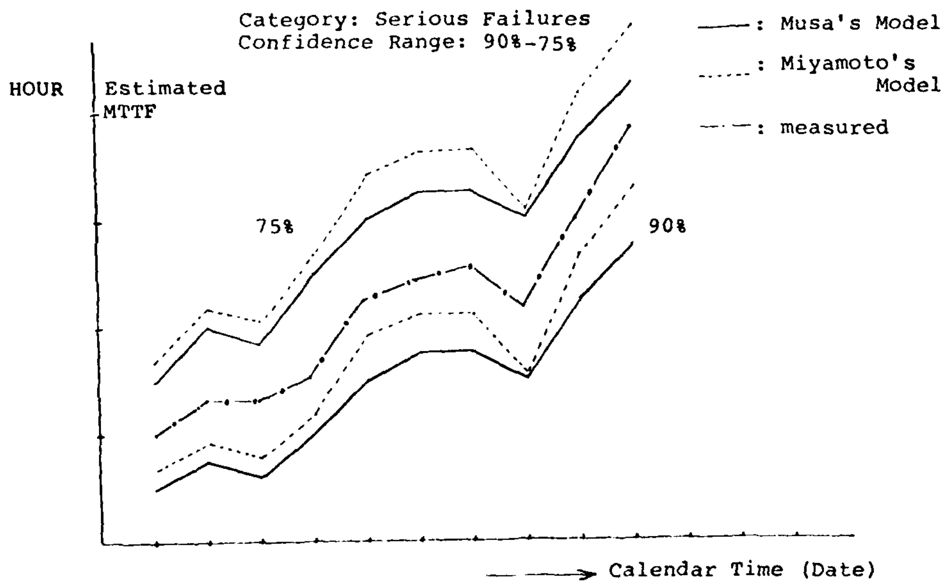


Fig. 6. Comparable Estimation of MTTF

ANALYSIS OF SOFTWARE ERROR MODEL PREDICTIONS AND QUESTIONS OF DATA AVAILABILITY

Alan N. Sukert

Rome Air Development Center (AFSC)
Griffiss AFB NY 13441

Abstract

During the period Aug 1974 to May 1978 a study to evaluate the accuracy of predictions of several models for predicting the error content and reliability of a software package against error data extracted from four large Department of Defense software development projects was undertaken by Rome Air Development Center (RADC). This paper will briefly describe the results of this empirical study for three such models, the Jelinski-Moranda, Schick-Wolverton and a modified Schick-Wolverton. Model predictions will be analyzed and general conclusions will be drawn as to model applicability. The data requirements for performance of such analysis will be discussed in lieu of the data RADC had available for this study and the data needed for such a study.

Introduction

The past several years have seen the formulation of numerous mathematical models for predicting the reliability and error content of a software system. These predictive tools were needed to permit better tracking of software developments by providing a software manager with more detailed information regarding the status of his development. Models formulated have ranged from the earliest ones that assumed an exponential distribution of time to detect errors, such as Shooman's, to more complicated ones such as Musa's execution-time model. These models have been experimentally tested against available software error data by the model developers and, whenever possible, comparisons of various models have been performed by model developers in order to demonstrate the applicability of each model.[6,9] However, criticism of this initial model testing has arisen due to the limited quantity and nature of the software error data available to model developers. This is especially true with respect to DOD software development projects, with their complex and unique, one-time applications in such areas as command and control and avionics.

To help develop a better knowledge about the applicability of these software reliability models, and to obtain better confidence in their predictions, RADC has been analyzing the predictions of several software reliability models against error data obtained during the formalized testing of several large DOD and NASA software developments. This paper will present the results of the RADC study and discuss the problems of data availability in performing this study. First, the models used will briefly be described. Next will follow brief descriptions of the four software development projects used in the study. Model predictions will be presented and analyzed with some conclusions offered as to model applicability. Finally, the problems of data availability will be addressed.

Model Discussion

The initial goal of this in-house study was to analyze as many software reliability models as possible, using as many software error data sets as possible. As the study evolved, it became very apparent that the limiting factor was data availability. Many models that would have been desirable to consider, such as the Shooman's exponential and Musa's models, were eliminated because the data available to RADC, which consisted of data extracted from Software Problem Reports (SPRs) that were filled out by the various contractors during the formal test phases whenever a software error was detected, was lacking in some of the needed categories such as CPU data. The models finally examined were the Jelinski-Moranda De-Eutrophication, Schick-Wolverton, Modified Schick-Wolverton, Jelinski-Moranda Geometric De-Eutrophication, and a Modified Geometric De-Eutrophication.

Predictions from these five models were first analyzed against data from a large DOD command and control project on a total project basis using Maximum Likelihood (MLE) estimates for model parameters.[1] Next, software error data from three additional DOD projects were analyzed against the three non-geometric models, since the three non-geometric models predicted the number of initial errors. Both MLE and Least Squares estimates for model parameters were used, and model predictions were obtained on a total project basis and also on an error criticality basis.[2] More recently, model predictions on a functional subsystem basis were obtained and analyzed.

The assumptions of the three non-geometric models used are given in Table 1, while the mathematical equations for the hazard function describing each of the three models are given in Table 2. For a more detailed description of the models the reader should consult Reference 1.

Table 1. Model Assumptions

Model	Assumptions
Jelinski-Moranda	1. The amount of debugging time between error occurrences has an exponential distribution with an error occurrence rate proportional to the number of remaining errors.
	2. Each error discovered is immediately removed, thus decreasing the total number of errors by one.
	3. The failure rate between errors is constant.
Schick-Wolverton	1. The amount of debugging time between error occurrences has a Rayleigh distribution.
	2. The error rate is proportional to the number of remaining errors and the time spent in debugging.
	3. Each error is immediately removed, thus reducing the number of errors by one.
Modified Schick-Wolverton	Same as Schick-Wolverton except for: 2. The error discovery rate is a constant during a time interval and is proportional to the number of errors remaining, the total time previously spent in debugging, and an "averaged" error search time during the current debug interval.

Table 2. Model Equations

Model	Hazard Function
Jelinski-Moranda	$z(t_i) = \phi[N - n_{i-1}]$
Schick-Wolverton	$z(t_i) = \phi[N - n_{i-1}]t_i$
Modified Schick-Wolverton	$z(t_i) = \phi[N - n_{i-1}][T_{i-1} - t_i/2]$

where: ϕ is the failure rate
N is the number of initial errors
n_i is the cumulative number of errors found through the i-th debugging interval
T_i is the cumulative time spent debugging through the i-th error

Notes: Hazard Function is the probability of an error occuring in a given infinitesimal time interval given that no error has occurred previously to that interval. The hazard function is related to the reliability R(t) and mean time to failure MTTF by the following:

$$R(t) = \exp[-\int_0^t z(s)ds] \quad (1)$$

$$MTTF = \int_0^\infty R(t)dt \quad (2)$$

Project Discussion

In this section a description of each of the four projects, from which the error data analyzed was obtained, is given. To maintain anonymity the projects are referred to as Projects 1, 2, 3 and 4.

Project 1

This project was a real-time control system for a land-based radar system written mostly in JOVIAL/J3, with the Executive and a few of the application modules written in Assembly.[14] The data obtained for this project was from the formal testing of all the project software, including the Executive. Formal testing began with Build Integration, where the modules were tested with the system executive and system data base. Upon successful completion of this testing a build was formed, which then was passed on to Acceptance testing. After completion of Acceptance testing the build entered Operational Demonstration, where a series of increasingly demanding mission profiles designed to exercise the system and evaluate its response were run. It is important to note that this system was a demonstration model, i.e. it was only designed to demonstrate that a system meeting the user requirements could be designed and built. It was never intended to become operational.

Project 1 software was developed using both top-down and bottom-up techniques and in a modular fashion. For example, module specifications were derived from the top-down starting with the system-level requirements. System integration was performed in incremental builds to check the interrelationships among the software modules and with the hardware. Dummy modules and drivers were used for testing those modules not part of a given build.

Project 2

Project 2 was a command and control system written in JOVIAL/J4.[12] The software was developed in a series of modifications with each modification governed by a separate set of requirements and developed independently. The software was developed functionally, i.e. the project was divided into work units responsible for different functions. Testing of each modification was conducted in five phases starting with Development testing by the development personnel to demonstrate specific functional capabilities, test data extremes, etc. Formal testing began after Development testing with Validation and Acceptance testing. Validation testing was performed by an independent test group at the subsystem level and demonstrated the approved software performance and requirements. Acceptance testing ran a subset of the Validation tests to demonstrate specific requirements. After this testing the software underwent final Integration testing by an independent group. This Integration testing demonstrated that the applications software correctly interfaced with the operating system and system support software. Data used in this study was from the formal testing of the Project 2 applications software only.

Project 3

Project 3 was a large command and control project written in JOVIAL/J4.[12] Structurally and procedurally, Project 3 was developed similarly to Project 2. However, the Project 3 software underwent an extra Operational Demonstration test phase in addition to the five test phases the Project 2 software underwent. The Operational Demonstration testing, which began after Integration testing, was designed to demonstrate the software in an operational environment using an operational timeline and operational data. The data obtained from this project was from the four formal test phases (Validation, Acceptance, Integration, Operational Demonstration) of the applications software.

Project 4

This project was a large avionics software application program written in JOVIAL/J3B and Assembly.[15] The software consisted of five major functional areas in the operational software and two in the simulation software. Testing of this software began with Module Verification testing performed by the developer of each module. Once this testing was finished, the module was released for formal testing. Formal testing began with Inter-Module Compatability testing where the software was checked against its functional requirements as a total unit, and which was done by the software development group. After completion of this testing the software system was given to an independent system test group for *Systems Validation testing*, where acceptance testing for quality control purposes was performed. The data obtained for this project was from the two formal test phases and is from both the operational and simulation software for the first two versions (called blocks) of the software system.

Table 3 contains a summary of the four projects.

Table 3. Project Characteristics

	Project			
	1	2	3	4
Language Used	JOVIAL/J3 Assembly	JOVIAL/J4	JOVIAL/J4	JOVIAL/J3B Assembly
Size (Lines of Code or No. of Mach. Inst.)	86780(J) 49900(A)	96931(J)	115346(J)	40640(J) 84065(A)
No. of Modules	109	173	249	
Operate Mode	Real-Time	Batch	Batch	Real-Time
Formal Testing	Build Integration Acceptance Operational Demonstration	Integration Validation Acceptance	Integration Validation Acceptance Operational Demonstration	Inter-Module Compatability Systems Validation

Model Results

The data used in this study was from SPRs and was restricted to those errors that resulted in a change to the software itself. The reason for this was that although unquestionably documentation errors are important and should be considered along with the other types of software errors, a confusion in the interpretation of the Project 2 and 3 non-software errors forced the arbitrary decision to eliminate all non-software errors, including documentation errors, to avoid confusion in interpreting model predictions. Opening dates were used instead of closing dates on the SPRs because of biases introduced in closing SPRs due to prioritizing and schedule demands.

The data was organized into errors per day and errors per week to see how the use of different time frames for the data affected model predictions. Since the data was subdivided functionally for all four projects, it was decided to use the date of the first SPR and the latest date that a subsystem began testing as two dates for model analysis to see if the start date of model prediction affected model predictability. All results will be given for both dates. Finally, operational data was available for Project 3 only. Thus all remarks made concerning model predictability for Projects 1, 2 and 4 are based on conversations with project developers and on relative comparisons.

All results will be presented in terms of the predicted number of remaining errors, i.e. the number of predicted initial errors minus the number of errors found to date. For notation purposes, the models will be denoted as follows:

Jelinski-Moranda (Maximum Likelihood):	JM
Jelinski-Moranda (Least Squares):	LJM
Schick-Wolverton (Maximum Likelihood):	SW
Schick-Wolverton (Least Squares):	LSW
Modified Schick-Wolverton:	MODSW

Table 4 presents a summary of the total project and summed criticality and subsystem predictions for the five models for Projects 1-4. Note that "---" indicates nonconvergence of the estimate equations for that particular model. Also, the numbers in "[ ]" are the number of errors found up through the end of formal testing, and the number in "( )" for Project 3 is the actual number of remaining errors.

Table 4. Total vs. Summed Predictions

Pj	Start Date	Model	Total System Pred		Criticality Pred		Subsystem Pred	
			Day	Week	Day	Week	Day	Week
1	1-02-73	JM	724	696	802	754	---	---
	[1853]	SW	26	156	37	116	---	---
		MODSW	14	13	13	12	43	38
		LJM	54	89	627	380	1326	678
		LSW	8821	1110	16477	4694	26995	11793
	3-06-73	JM	499	480	535	506	---	---
	[1769]	SW	14	108	41	73	---	---
		MODSW	8	7	8	6	23	20
		LJM	51	87	832	397	1524	620
		LSW	8814	984	16979	4824	28028	11828
2	10-14-71	JM	---	---	---	---	---	---
	[212]	SW	---	---	---	---	---	---
		MODSW	82	75	297	169	---	---
		LJM	140	57	372	188	633	373
		LSW	1993	750	3891	1677	5321	2594
	1-17-72	JM	72	66	---	---	---	---
	[189]	SW	39	56	---	---	---	---
		MODSW	5	3	15	9	---	---
		LJM	99	46	211	140	483	271
		LSW	1315	330	2237	560	4224	1902
3	6-01-73	JM	1288	1179	---	---	---	---
	[2191]	SW	955	1289	---	---	---	---
	(198)	MODSW	42	25	504	415	65	43
		LJM	88	23	390	143	814	419
		LSW	2155	685	2983	2023	11140	3550
	7-28-73	JM	233	198	---	---	---	---
	[1307]	SW	193	220	---	---	---	---
		MODSW	10	1	55	35	52	52
		LJM	81	27	338	139	645	340
		LSW	1322	505	2006	1166	5966	2355
4	5-22-73	JM	---	---			---	---
	[1877]	SW	---	---			---	---
		MODSW	7930	6328			---	---
		LJM	176	74			2710	1213
		LSW	9682	2202			28629	12379
	8-27-74	JM	650	591			---	---
	[1509]	SW	185	625			---	---
		MODSW	27	22			153	473
		LJM	116	63			739	488
		LSW	234	1103			10682	3250

Total Project Comparison

As can be seen from Table 4, there is a considerable difference in predictability when the different start dates for model prediction are used, and this difference shows up for all four projects. For example, for Projects 2 and 4 the JM and SW models fail to converge using the date testing started as the start date for model predictions, while these two models did converge when the date all models were ready for testing was used as the start date. Notice the significant drop in remaining error predictions in many instances between the use of the two different start dates. For instance, for Project 4 there is a factor of 100 drop in the remaining error predictions for the MODSW model when 8-27-74 is used, as opposed to 5-22-73, as the start date. The same is true for the MODSW remaining error predictions for Project 2, although here there is only a factor of 10 difference. However, there are instances, particularly for the LJM and LSW models, when there is no significant difference between the remaining error predictions using the two different start dates. Thus it would appear from the total project predictions that the difference in start dates affects the Maximum Likelihood parameter estimations more than the Least Squares parameter estimations.

The daily versus weekly model predictions do not offer as convincing a pattern. In most cases there does not appear to be a significant difference between using the day and week as the time interval, although the LJM and LSW models do in some cases show some nontrivial differences. For example, for Project 2 using the 10-14-71 start date the remaining error predictions for the LJM model were 140 using the day and 57 using the week. The LSW model predicted 1315 using the day and 330 using the week for Project 2 with a 1-17-72 start date. However, in some cases the "weekly" prediction was greater than the "daily" prediction. For example, the LJM model predicted 51 remaining errors using the day and 87 errors using the week for Project 1 with a 3-06-73 start date.

On a project versus project basis, clearly the LSW model predicts much higher values than the other models, while the MODSW model generally predicts lower values that the other models. From the actual remaining error count for Project 3 it is clear that those using 7-28-73 are much more accurate than those using 6-01-73, with some predictions being almost "too good". Since Project 2 and 3 are both command and control projects, one would expect that the same pattern of model predictability holds for both projects. From Table 4 it does appear that using 1-17-72 as the start date gives realistic predictions for all models except the LSW model, while for the 10-14-71 start date only the MODSW and LJM models give reasonable predictions. One would hope that the actual number of remaining errors was closer to the "1-17-72" predictions than the "10-14-71" predictions. Project 1, being a variation of a command and control project, would also hopefully give the same pattern of model predictability, and from Table 4 it appears that this is so. For Project 4, since it is an avionics development and thus significantly different from the other three projects, one would be interested in any differences in model predictability. From Table 4, one can note that the same general pattern appears with respect to the difference between the two start day predictions. However, it is interesting to note that the Project 4 predictions seem to be higher overall than those for the other three projects. The notable exception is the LSW model predictions for the 8-27-74 start date. This does suggest (at least for the limited data available) that the Maximum Likelihood parameter estimates might not be as accurate for avionics software predictions as the Least Squares estimates, while just the reverse holds for the Project 1-3 predictions (when the models converged). Obviously more testing is needed to verify this hypothesis.

Criticality and Subsystem Predictions

Note from Table 4 that no criticality data was available for Project 4. In order to understand the criticality predictions, it is important to note that the categorization of each error was made (by the developer) on the basis of the degree it was felt that error would impede the execution of a test case or prohibit demonstration of a requirement. As can be seen from Table 4, just as for the total project basis the model predictions are generally lower for the case when the date all modules are ready for testing is used as the start date than when the start of testing was used as the start date. The main exception is the LSW model predictions for Project 1. There is no three project pattern between the "day" and "week" predictions. For example, for Project 1 with a 1-2-73 start date, the criticality predictions for the JM, MODSW, LJM and LSW models for the "week" are less than or equal to the "day" predictions. However, the SW model predictions are higher for the "week" than for the "day". Note, however, that except for the Project 1 LSW predictions the "week" predictions for Projects 2 and 3 were lower than the "day" predictions.

From Table 4 it also appears that for the subsystem predictions, as was the case for the criticality and total project predictions, using the date when all modules are ready for testing gives generally more realistic predictions than using the date testing actually starts. Also, for the subsystem predictions the "day" versus "week" predictions showed a fairly consistent pattern for Projects 1-4 of the "week" predictions being lower than the "day" predictions, as was the case for the Project 1-3 criticality predictions, with the exception of Project 4 using the 8-27-74 start date.

Summarizing Table 4 we see that in most cases the "summed" predictions are greater than the "total" predictions. Note, however, that for Project 1 the SW criticality predictions total less than the "total" predictions for both start dates. Note also that the difference between the "summed" and "total" predictions is less for Project 1 than for the other projects. Overall, the MODSW model appeared to have the most consistent predictions, with the LJM and LSW models appearing to have the least consistent predictions. It appears that generally the criticality predictions are closer to the "total" predictions than are the subsystem predictions. However, this is not totally consistent, since for Project 3 the subsystem predictions are closer to the total than the criticality predictions for the MODSW model. Thus one can not draw any dominant patterns from this "total" versus "summed" analysis.

Conclusions

In presenting these results, attempts have been made to draw general conclusions about model predictions. Since no totally consistent patterns have evolved in most cases, general conclusions are difficult. However, since in most cases Project 1 appears to deviate from patterns that are dominant for the other projects, and since Project 1 was never intended to become operational while the other three were, one could eliminate Project 1 and make conclusions on the basis of model predictions for the other three projects. In formulating the following conclusions, I have done this only with respect to the subsystem and criticality predictions, since the pattern was so dominant for the Project 2-4 predictions.

Before stating any conclusions, however, a few words are needed about the nonconvergence of the JM and SW models (and in some cases the MODSW model). It had been hypothesized in [1] that the reason for this nonconvergence was the size of the data the models were applied against. The sizes of some of the data, particularly Project 2, somewhat negates this hypothesis. However, there does appear to be a pattern of nonconvergence for those data sets where the error density, i.e. the number of errors found per unit time, is sparse and uneven. This is especially true for Project 4. Thus it does appear that a significant factor in determining the convergence of the Maximum Likelihood estimates, since the Least Squares estimates always converged, is the rate of error detection. This would seem reasonable, since all three basic models implicitly assume a constant level of testing. A sparse uneven error detection density would certainly tend to negate this assumption. However, more research is needed to verify this.

From the above analysis, then, the following general conclusions can be drawn:

1. Clearly it is better to use the date all modules are ready for testing to begin model predictions than the date testing actually begins. This pattern was almost universally consistent among all the predictions.

2. For "command and control" projects such as projects 2 and 3, it appears that the Maximum Likelihood estimates, when they converge, give more reasonable and accurate estimates than the Least Squares estimates.

3. For "avionics" projects such as Project 4, it appears that the Least Squares estimates are more reasonable and accurate than the Maximum Likelihood estimates. However, this conclusion is somewhat suspect due to the nonconvergence problems for the Project 4 data.

4. For the criticality and subsystem predictions, it appears that using the week as the time interval gives more reasonable predictions than using the day as the time interval.

Data Availability

At this point the problem of data availability needs to be stressed. All of the conclusions made have been based upon work on data available to RADC. This data has some nice features, such as the availability in most cases of criticality designations and the ability to restructure the data based upon various classifications such as functional subsystems. However, the data does have some serious limitations. First, the data is historical in nature. Thus much of the information that one needs to perform a complete analysis of different software error prediction models, particularly CPU time data, was either totally lacking or of such a cumulative nature as to be unusable. Second, the definitions among different projects varied. For example, what one project called integration testing was not what another project meant by integration testing. How the various error categories were defined, as mentioned earlier for the Project 2 and 3 data, differed among the four projects. Also, the four projects were not consistent in what information could be provided. For example, no criticality data was available for Project 4. On another project from which error data was obtained but not used in this study, no information was available as to the opening date of the SPRS; only the closing date was available.

With the obvious shortcomings of the data available to RADC, it is more than likely that as more data does become available the conclusions drawn in this analysis may have to be slightly or totally altered. In addition, several patterns were noticed in analysis of the RADC data, such as the problem of model nonconvergence and the greater accuracy of the MLE or Least Squares estimates under certain conditions and for certain "types" of projects, that need more research. However, to perform this additional research requires both additional and more complete data. More data is also needed to perform more statistically valid tests for interpreting model accuracy.

Thus it becomes very clear that the limitations of the data available to RADC, and to most attempts to validate software error prediction models, requires more complete, and simply more, data. This implies several issues, though. First, precise definitions need to be made of the data elements that should be collected on software projects so that the maximum benefit can be obtained in terms of software error model analysis. In the case of DOD, this translates into the need for development of appropriate Data Item Descriptions (DIDs) that can be used in software procurements to require a contractor to collect the desired software error data in the desired formats. Second, data collection on on-going software developments must be initiated to provide complete and accurate data of the type necessary for model analysis. This data collection is also necessary to analyze projects that use, for example, modern programming practices to perform analysis of the benefits of such practices in a formalized statistical manner.

Last, and most important from an RADC viewpoint, there is the need for a centralized data base where researchers can obtain software data of the form necessary for his particular research. This data base must permit the specification of various categories and formats of software data on either a single project or cross project basis. The most difficult problem that was faced in the performance of this RADC study was the large time required to take the data from its original card image tape form and extract the necessary information in the required form, for input into the various models. A computerized data base for this type of information would have greatly expedited model analysis. Also, a computerized data base would facilitate storage of more data sets, and would help in specifying unique and consistent descriptors of the various data elements so that the "apples and oranges" problem can be eliminated. Finally, this data base will facilitate the use of software data for purposes other than research, most notably in tracking and managing large software development projects. RADC is currently developing a pilot Data Analysis Center for Software (DACS) that will service all kinds of software data and which will address the problems described above. Clearly such a center is needed for both research in software error prediction, and in providing better methods for monitoring software developments.

References

1. Sukert, Capt Alan N., "A Software Reliability Modeling Study", In-House Technical Report, RADC-TR-76-247, Aug 1976.
2. Sukert, Alan N., "A Multi-Project Comparison of Software Reliability Models", Proc. of AIAA "Computers in Aerospace" Conference, Oct 31 - Nov 2, 1977, pp. 413-421.
3. Moranda, P. and Jelinski, Z., "Software Reliability Research", McDonnell Douglas Astronautics Co., MDAC Paper WD1808, Nov 1971.
4. Moranda, P. and Jelinski, Z., "Final Report on Software Reliability Study", McDonnell Douglas Astronautics Co., MDC Report No. 63921, Dec 1972.
5. Moranda, P., "Probability-Based Models for the Failures During Burn-In Phase", Joint National Mtg ORSA/TIMS, Las Vegas NV, Nov 1975.
6. Moranda, P., "A Comparison of Software Error-Rate Models", McDonnell Douglas Astronautics Co.
7. Stucki, L., Moranda, P. et al, "Final Report - A Methodology for Producing Reliable Software, Vol. I", McDonnell Douglas Astronautics Co., MDC Report No. G6210, Mar 1976.
8. Wolverton, R. W. and Schick, G. J., "Assessment of Software Reliability", TRW Systems Group, TRW Software Series No. TRW-SS-72-04, Sep 1972.
9. Wolverton, Ray W. and Schick, George J., "An Analysis of Competing Software Reliability Models", IEEE Transactions on Software Engineering, Vol. SE-4, Mar 1978, pp. 104-120.
10. Lipow, M., "Maximum-Likelihood Estimation of Parameters of a Software Time-To-Failure Distribution", TRW Systems Group, Report No.

2260.1.9-73D-15(Rev 1), Jun 1973.
11. Lipow, M., "Some Variations of a Model for Software Time-To-Failure", TRW Systems Group, Correspondence ML-74-2260.1.9-21, Aug 1974.
12. Tal, Jacob, "Development and Evaluation of Software Reliability Estimators", University of Utah, Technical Report for Contract F42600-76-C-0315, Report No. SRL-76-3, Dec 1976.
13. Thayer, T. A. et al, "Software Reliability Study", TRW Systems Group, Final Technical Report, RADC-TR-76-238, Aug 1976.
14. Willman, H. E. Jr. et al, "Software Systems Reliability: A Raytheon Project History", Raytheon Co., Bedford Laboratories, Final Technical Report, RADC-TR-77-188, Jun 1977.
15. Fries, M. J., "Software Error Data Acquisition", Boeing Aerospace Co., Final Technical Report, RADC-TR-77-15, Apr 1977.

US ARMY COMPUTER SYSTEMS COMMAND

SECOND SOFTWARE LIFE CYCLE MANAGEMENT WORKSHOP

VI. ATTENDEE LIST

Paul R. ALLISON
U.S. Civil Service Commission
4685 Log Cabin Drive
Macon, Georgia 31204

M.A. ALLSHOUSE
USACSC
STOP C-70
Ft. Belvoir, Virginia 22060

W.A. (Gus) BAIRD
EES/RAIL
Electronics Rch. Building
Georgia Institute of Technology
Atlanta, Georgia 30332

Victor R. BASILI
Department of Computer Science
University of Maryland
College Park, Maryland 20742

Mert BATCHELDER
US Army Computer Systems Command
CSCS-POP
Ft. Belvoir, Virginia 22060

L. A. BELADY
IBM
P. O. Box 218
Yorktown Heights, New York 00598

G. BENYON-TINKER
Dept. of Computing & Control
Imperial College
180 Queens Gate
London SW1, England

Barry W. BOEHM
TRW Space Systems & Energy Corp.
One Space Park
Redondo Beach, California 90278

James E. BURNS
School of ICS
Georgia Institute of Technology
Atlanta, Georgia 30332

L.G. CALLAHAN
School of Ind. & Sy. Engineering
Georgia Institute of Technology
Atlanta, Georgia 30332

Randall CARRIER
Engineering Experiment St.
ERDA/TDA
Atlanta, Georgia 30332

Dr. Robert L. COOPER
Office of the Secretary of Defense
Room A-658 Pentagon
Washington, D.C. 20301

James COURTNEY
Industrial Management
Georgia Institute of Technology
Atlanta, Georgia 30329

Bill CURTIS
GE/ISP
Suite 200
1755 Jefferson Davis Highway
Arlington, Virginia 22202

Allan H. CURRY
AIRMICS
313 Calculator Building
Georgia Institute of Technology
Atlanta, Georgia 30332

K. Roscoe DAVIS
OBA Dept. of College of Business Administration
University of Georgia
Athens, Georgia 30602

Thomas G. DeLUTIS
OSU
8700 Duvall Street
Fairfax, Virginia 22031

Richard DeMILLO
Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Melvin DICKOVER
SofTech, Inc.
218 N. Lee, Suite 324
Alexandria, Virginia 23314

Dr. James ELSHOFF
General Motors Research Laboratory
Computer Science Department
Twelve Mile Road
Warren, Michigan 48090

Edward H. ELY
AIRMICS
313 Calculator Building
Georgia Institute of Technology
Atlanta, Georgia 30332

Philip ENSLOW
School of ICS
Georgia Institute of Technology
Atlanta, Georgia 30332

Kurt FISCHER
Computer Sciences Corporation
6565 Arlington Blvd.
Falls Church, Virginia 22046

John T. FITCH
AMCEE/Georgia Institute of Technology
Atlanta, Georgia 30332

Ross A. GAGLIANO
OR Group-EES
Georgia Institute of Technology
Atlanta, Georgia 30332

L. J. GALLAHER
EES
Georgia Institute of Technology
Atlanta, Georgia 30332

J. GEHL
Georgia State University
1290 Oxford Rd., N.E.
Atlanta, Georgia 30306

Clarence GIESE, Director
AIRMICS
313 Calculator Building
Georgia Institute of Technology
Atlanta, Georgia 30332

Amrit L. GOEL
427 Link Hall
Syracuse University
Syracuse, N.Y. 13210

Maurice HALSTEAD
(Purdue University)
228 Pawnee
Lafayette, Indiana 47906

Max HARRIS
(Naval Data Automation Command)
404 Tulip Court
Fredericksburg, VA 22401

L.T. HERRMANN
SHELL OIL
P.O. Box 20127
Houston, Texas 77025

Mary Anne HERNDON
Department of Mathematic Sciences
San Diego State University
San Diego, CA 92182

Larry C. HITCH
5949 Tyree Road
Winston, GA 30187

Dan HOCKING
AIRMICS
313 Calculator Building
Georgia Institute of Technology
Atlanta, GA 30332

John T. HOLLAND
(USAF)
54 Robinson Rd.
Lexington, MA 02173

George B. HOSLER
(USACSC)
1304 Crisfield Drive
Oxon Hill, MD 20021

Pei HSIA
University of Alabama
3300 O'Hara Drive
Huntsville, AL 35807

Doug HUEBNER
(GTE)
323 Marion St.
Glen Ellyn, IL 60137

Frank S. HUNTER
AIRMICS
Georgia Institute of Technology
Atlanta, Georgia 30332

James IRWIN
AIRMICS
313 Calculator Building
Georgia Institute of Technology
Atlanta, Georgia 30332

Dr. Larry A. JOHNSON
(LOGICON, Inc.)
18 Hartwell Avenue
Lexington, MA 02173

Hary S. KOCH
University of Rochester
Dewey Hall
Rochester, N.Y. 14607

Ken KOLENCE
Institute on Software Engineering
P.O. Box 637
299 California St. #203
Palo Alto, CA 94303

Klaus-Peter KOSCHEWA
AIRMICS-GIT
313 Calculator Building
Atlanta, Georgia 30332

Janet M. LATTYAK
IBM/ASTT
2401 Research Blvd.
Rockville, MD 20850

Richard LEBLANC
School of ICS
Georgia Institute of Technology
Atlanta, GA 30318

M.M. LEHMAN
Department of Computing and Control
Imperial College of Science & Technology
180 Queen's Gate
London SW72BZ England

Nate LIGOLS
(DOD)
12736 Rolling Brook Dr.
Woodbridge, VA 22192

M.A. LIPSCOMB
EES/SED
Georgia Institute of Technology
Atlanta, Georgia 30332

Bev LITTLEWOOD
City University
St. John Street
London ECIV4PB England

LTC(P) James E. LOVE
USACSC
ATTN: CSCS-VP
Fort Belvoir, VA 22060

Tom LOVE
GE/ISBD
401 N. Washington Street
Rockville, MD 20850

Thomas J. McCABE
5380 Mad River Lane
Columbia, MD 21049

J.A. McCALL
(General Electric Company)
979 Bucknam Avenue
Campbell, CA 95008

William McFADYEN
(GTE)
336 Drake
Bolingbrook, ILL

Sandra A. MAMRAK
Department of Computer & Information
Sciences
2036 Neil Avenue
The Ohio State University
Columbus, Ohio 43202

John H. MANLEY
Applied Physics Laboratory
Johns Hopkins University
Johns Hopkins Road
Laurel, MD 20810

Cecil E. MARTIN
Building 888
AFDSDC
Gunter AFS, Alabama 36114

Edith MARTIN
Computer Science & Technology
EES
Georgia Institute of Technology
Atlanta, GA 30332

Clair R. MILLER
Honeywell
7900 West Park Drive
McLean, VA 22301

Isao MIYAMOTO
c/o Computer Systems Research Labs
Room A-307, 1-1 Miyazaki Yonchome
Takatsu-ku, Kawasaki City
Kanagawa 213,
Japan

Barbara M. MOCK
Pentagon BD1033
USAMSSA
Washington, D.C.

Jerry MOHER
(USACSC)
1210 Priscilla Lane
Alexandria, VA 22308

John D. MUSA
Bell Telephone Laboratories
39 Hamilton Road
Morristown, N.J. 07960

J. David NAUMANN
761 BA
University of Minnesota
271 19th Avenue, South
Minneapolis, MN 55455

F.N. PARR
Department of Computing & Control
Imperial College
180 Queen's Gate
London SW7 England

G.L. PECKHAM
EES
Georgia Institute of Technology
Atlanta, Georgia 30332

Giora PELLED
AIRMICS
313 Calculator Building
Atlanta, Georgia 30332

D. Jason PENNEY
P.O. Box 35892
Atlanta, Georgia 30332

Ralph A. PETERSON, Jr.
(Georgia State University)
118 W. Paces Ferry Road, N.W.
Atlanta, Georgia 30305

John N. POSTAK
Doty Assoc., Inc.
416 Hungerford Drive
Rockville, MD 20850

Lawrence H. PUTNAM
Quantitative Software Management, Inc.
1057 Waverley Way
McLean, VA 22101

Morris W. ROBERTS
Department of Information Systems
Georgia State University
Atlanta, Georgia 30303

James E. RUSH
OCLC, Inc.
2223 Carriage Road
Powell, Ohio 43065

P.R. SANDER
P.O. Box 35622
Atlanta, GA 30332

George J. SCHICK
School of Business JSFW 403
University of Southern California
Los Angeles, California 90007

Louis SERNOVITZ
USACSC AIRMICS
Georgia Institute of Technology
313 Calculator Building
Atlanta, Georgia 30332

Alan D. SHERER
Computer Science Corporation
6022 Technology Drive
Huntsville, Alabama 35805

LTC. Joseph T. SHINE
AIRMICS
313 Calculator Building
Atlanta, Georgia 30332

James H. SHRAKE
Sperry Univac Corporation
1826 Summit Avenue
St. Paul, Minnesota 55105

P.J. SIEGMANN
School of ICS
Georgia Institute of Technology
Atlanta, Georgia 30332

Bobby B. SIMMONS
(USACSC)
5209 Light Street
Springfield, VA 22151

LuAnn SIMS
785 Ashland Avenue
(Georgia Institute of Technology)
Atlanta, Georgia 30307

Robert W. SMART
USACSC Support Group
Fort Lee, VA 23801

Shelly SMITH
Georgia Institute of Technology
Box 37064
Atlanta, Georgia 30332

John STAUDHAMMER
USARO
P.O. Box 1221
RTP, North Carolina 27709

Barb STEWART
Honeywell, Inc.
MN17-2375
2600 Ridgway Parkway, N.E.
Minneapolis, Minnesota 55413

Harold S. STONE
Electrical & Computer Energy Department
University of Massachusetts
Amherst, Massachusetts 01002

Leon STUCKI
Boeing Computer Services, Inc.
Mail Stop 3W-52
P.O. Box 24346
Seattle, Washington 98124

Alan N. SUKERT
RADC/IRIS
Griffiss AFG, New York 13440

Dr. Robert TAUSWORTHE
Jet Propulsion Laboratory
4800 Oak Grove Drive
238-640
Pasadena, California 91103

Robert THIBODEAU
General Research Corporation
307 Wynn Drive
Huntsville, Alabama 35805

Rick THOMAS
(Georgia Institute of Technology)
1172 Kendrick Road
Atlanta, Georgia 30319

W.M. UNDERWOOD
School of Information & Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30307

Dr. Joseph E. URBAN
Program Management Division
ATZ HTD-P
Directorate of Training Development
Army Signal Center
Fort Gordon, Georgia 30905

Cyprian VICARS
AIRMICS
313 Calculator Building
Atlanta, Georgia 30332

Harrison M. WADSWORTH
School of ISYE
Georgia Institute of Technology
Atlanta, Georgia 30332

Maj. Alfred W. WALEA
U.S. Army, Hessa
Building 200
Fort Sam Houston, Texas 78234

Dr. Claude WALSTON
IBM Federal Systems Division
10215 Fernwood Road
Bethesda, Maryland 20034

Thomas E. WATKINS
AFDSDC/DMT
Gunter AFS, Alabama 36114

Gerald M. WEINBERG
Ethnotech, Inc.
RFD #2
Lincoln, Nebraska 68505

F.E. WILLIAMS
Industrial Management
Georgia Institute of Technology
Atlanta, Georgia 30329

Dr. Ray W. WOLVERTON
TRW Defense & Space Systems - DSSG
Office of Software Research & Technology
R2-1086
One Space Park
Redondo Beach, California 90278

Donovan YOUNG
ISYE
Georgia Institute of Technology
Atlanta, Georgia 30332

Marvin ZELKOWITZ
University of Maryland
Computer Science Department
College Park, Maryland 20742

ATE
LME